8-1-2011

Investigating data throughput and partial dynamic reconfiguration in a commodity FPGA cluster framework

Nicholas Palladino

Investigating Data Throughput and Partial Dynamic Reconfiguration in a Commodity FPGA Cluster Framework

by

Nicholas L. Palladino

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Computer Engineering

Supervised by

Associate Professor Dr. Muhammad Shaaban
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2011

Approved by:

Dr. Muhammad Shaaban, Associate Professor
Department of Computer Engineering

Dr. Marcin Lukowiak, Assistant Professor
Department of Computer Engineering

Dr. Sonia Lopez Alarcon, Assistant Professor
Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Investigating Data Throughput and Partial Dynamic Reconfiguration in a Commodity FPGA Cluster Framework

I, Nicholas L. Palladino, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Nicholas L. Palladino

Date

Dedication

This thesis is dedicated to my parents. Without their support and guidance, I would not have been able to reach this point in my academic career.

Acknowledgments

I would like to thank my primary faculty advisor Dr. Shaaban, for his guidance and support, as well as the rest of the committee for feedback. I would also like to thank the Computer Engineering department for providing the cool hardware to play with. Also, a big thanks to Charles Gruner and Joe Walton for lending their System Administrator expertise.

Abstract

Investigating Data Throughput and Partial Dynamic Reconfiguration in a Commodity FPGA Cluster Framework

Nicholas L. Palladino

Supervising Professor: Dr. Muhammad Shaaban

There are many computational kernels where parallelism can be exploited in application specific hardware, yielding significant speedup over a general purpose processor based solution. Commodity cluster computing technologies have been combined with FPGA coprocessors, resulting in even greater performance capability through the exploitation of multiple levels of parallelism. One particularly economic solution both in terms of cost and power consumption is to cluster hybrid FPGAs with commodity network interconnects. Hybrid FPGAs combine embedded microprocessors with reconfigurable hardware resources on a single chip offering lower power consumption and cost compared to a traditional I/O bus FPGA coprocessor solution. While there is a lot of promise in using commodity hybrid FPGAs in a cluster configuration, the design flow and performance characteristics of such systems are currently a limiting factor to the range of applications that could benefit from such a system.

The contribution of this thesis is a framework for clustering commodity FPGAs which integrates high speed DMA data transfers with a flexible FPGA resource sharing scheme enabled through partial reconfiguration. The framework includes an embedded Linux operating system, with a custom device driver to manage data transfers and hardware reconfiguration. User space tools for cluster computing including ssh and MPI are deployed allowing tasks to be split among nodes in the cluster. Performance analysis is performed with a homogeneous cluster composed of four Virtex-5 FXT based FPGA boards. The results demonstrate the advantages over previous work in terms of data throughput and

reconfiguration, as well as promote future research efforts.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, cluster computing with a focus on the strength of FPGAs is explored. Motivation for the work performed in this thesis is discussed. The contribution of this thesis is stated and an outline of the paper is presented.

1.1 Motivation

High-Performance computing is a field driven by advances in parallel computer architecture. In the 1970's, vector processor based machines such as the CRAY-1 supercomputer dominated the field [1]. These expensive, custom designed machines were superseded by clusters of Commodity Off The Shelf (COTS) microprocessors which offered better price/performance ratios. Many modern supercomputers continue to use clusters of microprocessors, however, clusters of nodes with specialized accelerators are increasingly being investigated and deployed. In fact, three out of the top ten supercomputers on the November 2010 Top-500 list use GPU accelerators. While GPUs have been successful in supplementing the floating point performance of microprocessors, applications that do not benefit from high speed floating point operations will not see any benefit. An alternative line of research is to augment microprocessor nodes with FPGA hardware accelerators. FPGAs are good for computations which can take advantage of bit level parallelism, some examples being signal processing, image processing, cryptanalysis, and bioinformatics [5]. FPGAs will also soon be competitive with CPUs in terms of price/FLOP while having less power consumption [2]. These characteristics have led to FPGAs being used in various

supercomputers including commercial offerings from SRC [3] and Cray [4].

FPGAs coprocessors have been coupled at various levels to the host CPU. Common methodologies include through a system I/O bus such as PCI/e or directly to the system bus via an addition CPU socket. The tightest coupling currently available places one or more embedded microprocessors in the same chip as the FPGA resources. These Hybrid FPGA chips offer reduced power consumption over a traditional CPU host system and are available on commodity development boards manufactured by Xilinx. The cost is competitive with a high performance microprocessor cluster node. Because of the advantages of Hybrid FPGA based development boards, multiple research efforts have been carried out demonstrating the viability of using the boards in a cluster configuration [6] [7]. These previous efforts have demonstrated the potential of commodity Hybrid FPGA clusters, but do not take advantage of modern Hybrid FPGA features including partial reconfiguration and high speed DMA data transfers of data from the CPU to hardware.

Thus, one area of research that has yet to be performed is to integrate runtime reconfiguration of hardware accelerators within a commodity FPGA cluster framework utilizing Hybrid FPGAs. With new partition based partial reconfiguration technology from Xilinx, it is possible to create a flexible clustering framework allowing multiple accelerators to be shared within a single FPGA and across multiple FPGAs. The basic framework infrastructure will build upon existing work [6] which demonstrates the viability of running Linux on the embedded processors and using MPI for communication across and within FPGA nodes. In addition, the viability of silicon DMA controllers in the Virtex-5 FXT architecture will be examined for data transfers between the host CPU and hardware accelerators.

1.2 Contribution

This thesis will contribute a framework for clustering commodity FPGAs which expands upon previous research efforts. Data throughput improvements will be made by performing DMA transfers between the embedded CPU and hardware accelerators. A flexible FPGA

resource sharing scheme will be enabled through partial dynamic reconfiguration. A Linux device driver and custom hardware will be designed to enable these features. The design flow for deploying a Loopback accelerator will be studied. This includes both hardware and software design targeting the framework. Performance analysis will be performed to identify critical performance characteristics that serve as an evaluation of the framework and give a good indication of what future applications could potentially be deployed to the platform and benefit from what it offers.

1.3 Thesis Organization

Chapter 2 discusses the enabling technologies allowing the work in this thesis to be performed, namely cluster computing, and FPGA technology. Chapter 3 reviews related research and how this thesis builds on previous works. This includes previous commodity FPGA cluster efforts, data throughput techniques in Hybrid FPGAs, and partial dynamic reconfiguration research. Chapter 4 describes the hardware and software framework developed to meet the design goals of this thesis. Chapter 5 describes performance characteristics of the developed framework and how it is an improvement over previous works. Chapter 6 provides a conclusion and offers suggestions for future research.

Chapter 2

Background Information

This chapter provides an overview of the enabling technologies and foundational concepts which allowed this work to be completed. Cluster computing ideologies and technologies are discussed. A history of FPGA technologies as they relate to this thesis is given.

2.1 Cluster Computing

Cluster computing emerged as the successor to traditional vector supercomputers of the 70's and 80's. This emergence came about from advances in consumer microprocessor technology and message passing software to facilitate communication between nodes. Microprocessors were created with features including multiple issue, pipelining, and an increased number of functional units which significantly boosted the performance of consumer microprocessors, making them a viable building block of a new type of supercomputer. The other critical piece to facilitate cluster computing was the development of the Parallel Virtual Machine (PVM) software in the late 80's. PVM was an open source tool that provided libraries for enabling networked computers to communicate and coordinate operations. Stemming from these innovations was the development of the first Beowulf cluster in 1994. A Beowulf Cluster is a scalable cluster using commodity hardware connected on a private network using open source software [9].

In 1994, Message Passing Interface (MPI) was developed to address shortcomings of

PVM. MPI is the current standard for message passing between clustered computer systems. It is an open standard and has various implementations across many hardware platforms. It supports many different interconnects and is the de facto message passing interface used in clusters. In 1997, MPI-2 was introduced which expanded the standard by adding dynamic process creation, one-sided operations, and parallel I/O [10]. MPI is widely used and the APIs are familiar to many different developers making it a popular standard for parallel programming on clusters.

Today, clusters dominate the field of High Performance Computing. From the June 2011 Top 500 list, 82.2% of the super computers are clusters. It is clear that the cluster architecture has proven to be successful. The driving force behind the evolution and technological progression of clusters has been the need to accommodate applications which need to become tractable within a certain time constraint or perform more detailed computations for existing applications. Clusters are able to solve complex problems by taking advantage of parallelism in the target applications. When the problems can be effectively broken into smaller chunks that can be executed in parallel, they will benefit from running on a cluster. There are different degrees to which a problem can be broken down, this measure is called the degree of software parallelism. At the highest level is task parallelism in which multiple threads of execution are possible. There are also lower levels of parallelism such as data parallelism where operations are performed on independent data. There is instruction level parallelism where independent instructions can be executed simultaneously. These lower level types of parallelism are exploited in modern CPU architectures.

Even with CPUs exploiting low level parallelism in software, there is room for improvement. More recent trends have been to augment cluster nodes with special purpose hardware to improve performance further for certain classes of problems. Different techniques have been employed including using special purpose parallel processors such as the Cell, or adding coprocessors such as GPUs or FPGAs. While each of these architectures have their own successes and limitations, more research is continually being performed to

see how far each architecture can be pushed. As long as there are applications to drive progression, there will be technological innovations to meet the need.

2.2 FPGA Technology

FPGA technology has advanced a lot since it's introduction in 1985 by Xilinx. FPGAs offer reconfigurable hardware resources in the form of configurable logic blocks connected through programmable routing. In an effort to improve performance is key areas, silicon IP implementations have been increasingly placed along side the reconfigurable fabric. Example of such hard IPs include ethernet controllers, DMA controllers, DSP elements, and embedded microprocessors in the case of Hybrid FPGAs. The following sections discuss different areas of FPGA technology including Hybrid FPGAs, Partial Dynamic Reconfiguration, and FPGA Design Tools.

2.2.1 Hybrid FPGAs

Hybrid FPGAs combine one or more silicon embedded processors with FPGA resources on a single chip. Currently, these devices offer the closest coupling between FPGA and CPU. This means reduced power consumption and good throughput as the interconnect is on chip. These FPGAs are typically used for embedded applications as they pack a lot of computational capability into a low power package.

Xilinx has had four generations of such devices. The first was the Virtex II Pro released in 2002 [11]. This architecture supported up to four PowerPC405 cores operating at up to 300MHz. The Virtex-4 FX was released in 2005 [12]. This FPGA included an improved PowerPC 405 processor capable of operating at up to 450 MHz. It also added the Auxiliary Processor Controller Unit (APU) to allow the processor to execute custom instructions in the FPGA fabric. In 2007, the Virtex-5 FXT was released [13]. This is the architecture used in this these. These device provide up to two PowerPC 440 processors operating at up to 500 MHz. Many innovations were also made to the way the processor connects

with the rest of the system. A 5x2 crossbar interconnect mechanism was added to provide simultaneous access to memory and I/O. Additionally four full duplex DMA channels were added to facilitate high throughput data transfer between the processor and FPGA fabric. Xilinx announced the ZYNC-7000 family of devices to be released in 2011 [14]. The ZYNQ platform will have a dual-core Cortex-A9 processor with double precision floating point engines. The device will contain a high-bandwidth AMBA4 Advanced Extensible Interface (AXI4) interconnect allowing multi-gigabit transfers between the processor and programmable logic.

2.2.2 Partial Dynamic Reconfiguration

Partial Dynamic Reconfiguration is a technology that allows part of the FPGA to be reconfigured while the rest of the FPGA operates undisturbed. This capability has many benefits. A single FPGA can be used for much more as different hardware designs can be timeshared without requiring the device to be completely reprogrammed and reset. This means space can be saved on the FPGA by enabling dynamic swapping of modules rather than having them all statically implemented. As a result, power consumption can be reduced since bitstreams can be stored in memory until need rather than being implemented on the device. Reconfiguring only a part of the FPGA also means less time is required to perform the configuration. Additionally, synthesis times could be reduced by implementing only the dynamic portion of the design.

Xilinx traditionally has been the only manufacturer to continually offer FPGAs with improved partial reconfiguration capability. Xilinx is up to its forth generation of partial reconfiguration technology. The first Xilinx device to officially support partial reconfiguration was the XC6200 announced in 1995. The XC6200 was also the first FPGA to offer optimizations for connecting to a host processor bus, making it an ideal device for coprocessing. The second generation partial reconfiguration technology is called difference-based partial reconfiguration. This technology was introduced in 2000. Difference-based PR is used for making small changes to a design like changing LUT equations using the Xilinx

FPGA Editor. All Virtex devices are supported, however It is not useful for swapping out a large portion of the design. The third generation technology introduced the modular design flow. In this flow, regions of partial reconfiguration logic could be specified and is connected to the static portion of the design through bus macros. The modular reconfiguration approach has some shortcomings including custom build steps, and each PR region must be a slice the height of the FPGA fabric.[16] The fourth generation tools use a Partition-based flow. The new Xilinx 12.1 Design Tools include a flow that makes creating a system utilizing partial reconfiguration for processor peripherals easier. The new flow uses proxy logic which is a single LUT1 element placed for each pin between the static design and reconfigurable partition [29]. Reconfigurable partitions can be laid out in PlanAhead and integrated with an EDK project.

2.2.3 Design Tools

Tools for FPGA design and synthesis are still very proprietary due to the tight coupling to the hardware they are used on. Thus, each manufacturer has their own proprietary design tools. Xilinx has various software packages available depending on the type of system that is being targeted. In this thesis, the Xilinx Design Suite 12.1 software was used. Many different applications and utilities are included with the Xilinx software. Each of the applications used in this work will be described to present some background on how they will be used.

First is Xilinx Platform Studio (XPS). This software allows a user to generate an embedded system targeting a specific FPGA architecture, in the case of this work, a Virtex-5 FXT device. Through XPS, the Base System Builder (BSB) utility can be used to generate a baseline configuration. BSB provides a step-by-step wizard that guides a user through creating a Microblaze or PowerPC based system. The listing shown in Table 2.1 gives the configuration used in BSB to generate the base system. The Processor Local Bus (PLB) is the system bus through which various I/O peripherals are connected to the processor. With a CPU frequency of 400MHz, 100MHz is the maximum allowed bus clock rate for this

system. The XPS project file is included with the thesis materials for future researchers to use and modify as they see fit.

Component	Parameters
Processor Type	PowerPC 440
Processor Clock Frequency	400 MHz (Virtex-5)
Processor Cache	64 KB (32KB Data + 32KB Inst)
PLB Clock Frequency	100 MHz
UART Controller	XPS UART16550
Hard Ethernet MAC	Scatter-Gather DMA (RGMII)
DDR2 RAM	PowerPC Memory Controller
Compact Flash	XPS SysAce

Table 2.1: BSB Common Configuration Options

PlanAhead is a tool used for floor planning a design and generating reconfigurable partitions to allow partial dynamic reconfiguration. A PlanAhead project supporting partial dynamic reconfiguration was created. The use of the partial dynamic reconfiguration requires a special license from Xilinx beyond the license necessary to run PlanAhead. PlanAhead takes a netlist generated from XPS. From there, the reconfigurable partitions can be created for blackbox components in the netlist. After these partitions are created, individual accelerator netlists can be targeted at each partition. After this is done, the static and partial bitstreams can be generated. The PlanAhead project file is included with the thesis materials for future researchers to use and modify as they see fit.

ISE is a tool used to synthesize hardware designed. In this work, individual accelerators are synthesized to netlists in ISE. From there, they are brought into the PlanAhead project and deployed as partial bitstreams to the framework. A testbench simulating the accelerator framework developed in this thesis was created for use with Modelsim. Modelsim is a simulation utility used to verify hardware designs. The designs are verified in Modelsim using the testbench before being synthesized and deployed.

Chapter 3

Related Research

This chapter discusses related research and how this thesis makes a contribution beyond what has previously been done. This thesis primarily builds on a previous thesis entitled, Scalable Framework for Heterogeneous Clustering of Commodity FPGAs. Since this work sets a foundation for the work to be performed in this thesis, it will be given significant attention in this chapter. The successes and shortcomings of the framework are examined and design decisions made to improve the framework are motivated based on other bodies of work.

3.1 Commodity FPGA Clusters

The idea of a beowulf cluster using commodity FPGAs is not new. In fact there have been multiple research efforts in the area, some still ongoing. The following sections will look at the previous works and identify areas where the proposed work improves on those efforts.

3.1.1 Scalable Framework for Heterogenous Clustering of Commodity FPGAs

In [6], a framework was developed which enabled the use of commodity hybrid FPGAs with custom Hardware Accelerator Units (HAUs) to be used in a heterogenous cluster configuration. Previously, there was no established platform which satisfied these goals. The framework allows HAUs implemented in FPGA logic to communicate as independent cluster nodes via MPI. Network communication is handled by the embedded PowerPC processor on the Hybrid FPGA running Linux. Figure 3.1 shows a diagram of the cluster network topology. Each FPGA can have as many HAUs as will fit on the device. Each

HAU is managed by a process running on the PowerPC.

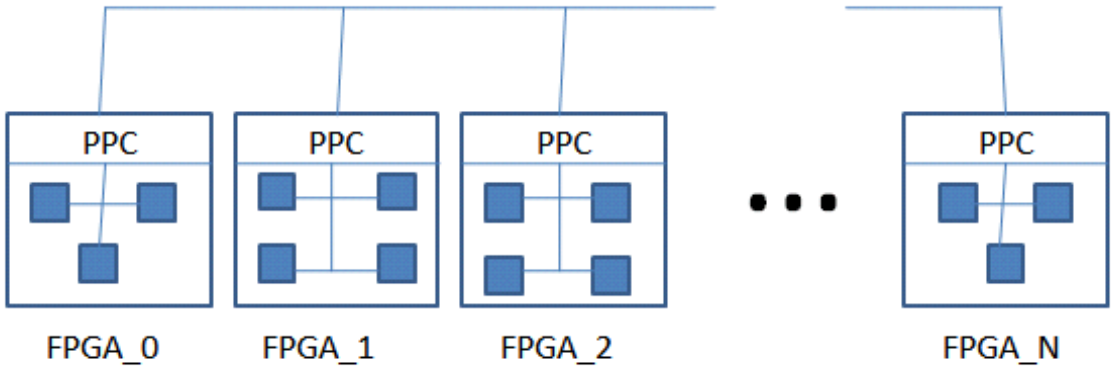

Figure 3.1: Framework Cluster Topology [6]

The high level design goals of the framework were that it was easily programmable, utilized independent HW/SW co-design, had minimal framework overhead and was scalable across FPGAs. Programmability was handled by developing around standard APIs. MPI was used for interprocess communication while typical character device file operations, fopen, fwrite, fread, and fclose were used for communicating with HAUs. An application agnostic interface was established for software and hardware developers to target. Software has to send data in a way that the hardware is expecting, however there are no other implementation specific requirements. Framework overhead and scalability were combatted by utilizing existing technologies and developing a flexible software/hardware interface.

Figure 3.2 shows the application stack going from a user MPI Application down to the application HAUs. The software runs on an embedded Linux operating system. The root file system was created and cross-compilation was performed using the DENX Embedded System Development Kit. The OpenMPI implementation of the MPI standard was installed on the root file system. Additionally, OpenSSH and OpenSSL were installed to accommodate secure login capabilities. The root file system was copied to an Ext2 partition on a compact flash card and used for each FPGA node in the cluster. The interface between

software and hardware was provided thorough the use of FIFO queues. FIFOs allow a flexible interface between HAUs and the Processor Local Bus (PLB) which they are attached to. A 32-bit configuration of the PLB is supported by Linux drivers and was used in this work. Each HAU has a write and read FIFO attached to the PLB. Data from the processor is written out over the PLB into the write FIFO of a HAU. The HAU then processes the data from the FIFO and writes to the read FIFO. The processor will then request to read data back from the HAU's read FIFO. Additionally, interrupt generation is supported so that the device driver can sleep until the HAU indicates it is done in the case of long lasting hardware processing. Figure 3.3 shows a diagram of the bus and HAUs along with out typical system peripherals.

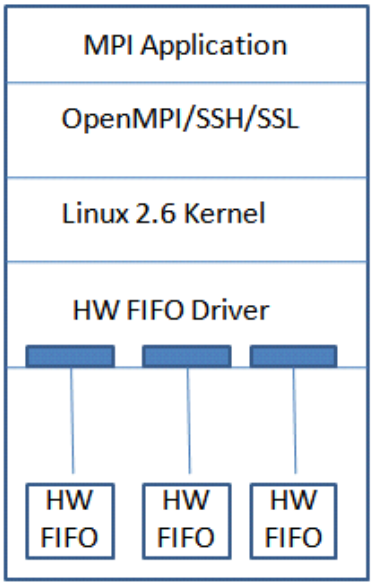

Figure 3.2: Application Stack spanning Hardware and Software Design [6]

While providing a nice and easy to use hardware/software abstraction, the framework suffered some shortcomings. The FPGA devices used in the work offered features that could greatly improve the usability of the developed framework. The first minor issue is that although some of the devices had gigabit ethernet capability, others did not, so the feature was not enabled. The next more serious issue was the limited throughput between

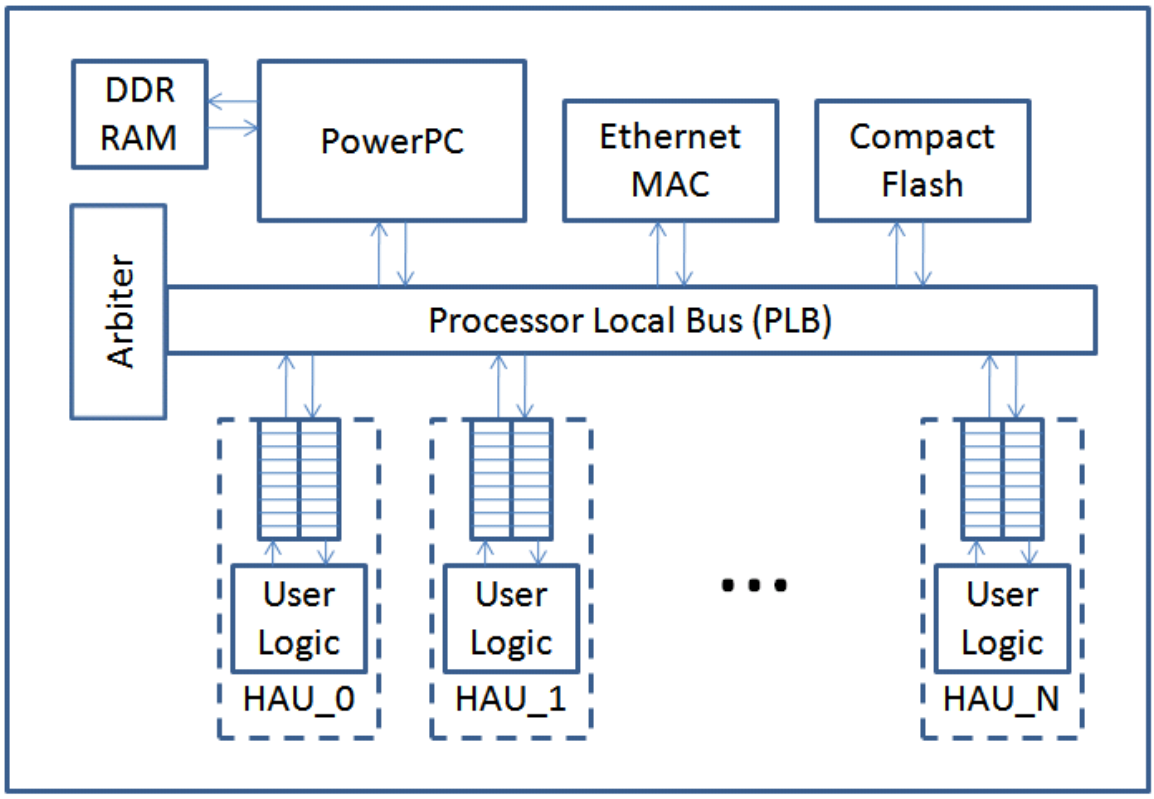

Figure 3.3: Framework Accelerators [6]

the processor and HAUs over the PLB. This is perhaps the biggest performance bottleneck that would prevent a wider range of applications from being successful on the platform. Finally, the deployment of new HAUs required recompiling the Linux kernel, and rebooting with the new kernel and updated bitstream. This is very undesirable in terms of deployment process and the ability to share hardware between users.

3.1.2 Reconfigurable Computing Cluster

The Reconfigurable Computing Cluster Project has created one of the largest commodity FPGA based clusters, which is made up of 64 Xilinx ML-410 development boards [8]. The goal of the project is to investigate the feasibility of using commodity FPGAs in a cluster that will scale to the PetaFLOP level [7]. The RCC infrastructure differs significantly from the proposed work in this thesis. The goal of the RCC is to allow a user to login through the

head node and gain complete access to an FPGA, blocking any other users from using that board [28]. This is accomplished by using multiple networks, a gigabit ethernet network, a custom point-to-point network, and a USB network. In contrast, the proposed work will allow multiple users to share a single FPGA by splitting the FPGA resources into multiple discrete reconfigurable blocks. Only a gigabit ethernet network will be used, simplifying the design, and saving costs.

3.2 Hybrid FPGA Data Transfer Methodologies

3.2.1 Hybrid OS

One particularly comprehensive work studying different transfer methodologies was performed on the Virtex II Pro platform running Linux [15]. In this work, a hardware framework was designed to facilitate data transfers between the PowerPC processor and accelerators as well as between accelerators. Various methods of device driver access to the hardware were examined. In the study, User mapped DMA transfers were shown to have the best throughput if the transfer size was larger than 160 Bytes. User mapped DMA maps DMA buffers to the user's virtual address space. This eliminates extraneous copying of data from user space buffers to kernel space and back. This design has some complications however that make implementation difficult for use with multiple accelerators. Direct mappings were also explored where accelerator memory is mapped to user space both in cacheable and non cacheable configurations. It was shown that cacheable directing mapping could result in faster transfers than User Space DMA for transfers up to 512 Bytes. Uncachable Direct Map was shown to be the slowest transfer methodology tested. This method is equivalent to writes to a PLB slave peripheral as the peripheral address range in in non-cacheable memory. This thesis will use a User Space DMA approach which will improve throughput while eliminating the complexity of a User Mapped DMA implementation and still achieving significant speedup.

3.3 Partial Dynamic Reconfiguration Methodologies

Partial Dynamic Reconfiguration has been an active area of research and is becoming more popular as tools and FPGA architectures continue to improve. Historically, partial reconfiguration capabilities have been limited and hard to work with [16]. One recent work [32] has used the Partition based partial reconfiguration flow from Xilinx. This work was developed at this same time as this thesis, and thus are the first two works to examine the partition based partial reconfiguration flow in cluster frameworks. Additionally, methods for utilizing the ICAP for reconfiguration have been examined and motivate the approach used in this work.

3.3.1 ArchES-MPI

In [32], partial reconfiguration capabilities were added to the ArchES-MPI framework. ArchES-MPI is a framework which provides a communication abstraction layer that enables point to point communication between X86 processors, embedded processors, and hardware accelerators. ArchES-MPI is an expansion on TMD-MPI which provides a subset of the MPI standard that can be implemented in FPGA hardware. The work uses the Xilinx Partition based flow to facilitate creating reconfigurable partitions. The long term goal of the work is to implement a generic framework that allows hardware designers to create generic template platforms that follow a parallel programming model easier for software developers to understand. In the framework, state store and restore of reconfigurable modules is not performed as in other work [17]. Rather, it is left to the user to decide what to store and restore using the message passing interface. In ArchES-MPI, there are software ranks running on processors and hardware ranks running on hardware engines. Each hardware rank can have multiple reconfigurable modules. This varies from the work proposed in this thesis since there are only software ranks in from which reconfigurable modules can be swapped in and out.

The design flow for deploying accelerators in the framework is split into two flows. The

first is to partition the design into static and reconfigurable regions. The second part of the flow is to use Xilinx EDK/ISE 12.1 to implement and synthesize the system design. Then PlanAhead is used to configure reconfigurable partitions. Users implement hardware accelerators in HDL and synthesize them. Once netlists have been generated after synthesis, they are brought into PlanAhead. Once in PlanAhead, the netlists are targeted to reconfigurable partitions. The partial bitstream for each accelerator is generated after place and route of each accelerator is performed. The last step is to run a script that compiles the MPI code for all of the processors used in the system. This thesis follows a similar deployment of accelerators, using PlanAhead to generate partial bitstreams from accelerator netlists.

Reconfiguration in the framework is facilitated through the following function call ARCHES_MPI_Reconfig(). The function takes three parameters, a partial bitstream filename, FPGA board number, and FPGA number. This allows an application to reconfigure an accelerator on a target board. The bitstream contains information regarding which accelerator configuration memory will be overwritten. This approach differs from the approach taken in this thesis. Rather than having an API call that handles reconfiguration behind the scenes, reconfiguration in this work is handled through an ioctl call in the device driver. There was no need to abstract reconfiguration of accelerators further.

One limitation of ArchES-MPI is that the full MPI standard is not implemented. One example of a missing feature is dynamic process creation. This means that any existing code that relies on this capability would have to be re-architected to run on the framework. The approach taken in this thesis is to use a software OpenMPI implementation running on an embedded processor which implements the full MPI standard. This means that all MPI features are available and the user must then only worry about efficiently using hardware resources.

3.3.2 Partial Reconfiguration Speed Investigation

Xilinx Virtex 5 FPGAs have six configuration interfaces for bitstream programming being Serial, SPI, BPI, SelectMap, JTAG, and ICAP [24]. The Internal Configuration Access Port

(ICAP) is unique because it enables an embedded microprocessor to reconfigure parts of the FPGA at runtime. In addition, there is the Xilinx XPS HWICAP IP which provides a Processor Local Bus (PLB) interface to the ICAP [21]. There is also a Linux device driver for the HWICAP in the Xilinx Linux kernel tree. Thus, the interface to the ICAP from the PowerPC core running Linux can be established by building on existing work. It is important to note however, that there has been research to determine the fastest ways to reconfigure an FPGA using the ICAP interface from an embedded microprocessor.

There has been one particularly comprehensive work investigating performance of various design architectures for utilizing the ICAP [31]. A number of different designs were analyzed for fastest reconfiguration time. The analysis was performed on a Xilinx ML405 development board with a Virtex-4 FX20 FPGA. The fastest design had an average reconfiguration speed of 332.1 MB/s which is approaching the theoretical maximum rate of 400 MB/s of the ICAP interface (32-bit, 100MHz). The design was called BRAM_HWICAP. In the BRAM_HWICAP design, a dedicated BRAM block is used to store bitstream data. The BRAM has to be big enough to hold the entire bitstream. The BRAM acts as a cache in which bitstreams can be downloaded via the PLB prior to reconfiguration time. When a reconfiguration is desired, it will be faster since the time to transfer the bitstream from memory to the HWICAP is no longer needed. This case was demonstrated to work well when the PR region is small and the fastest reconfiguration time is needed. However, for larger bitstreams, BRAM resource utilization becomes prohibitively high.

Additionally, a DMA design, DMA_HWICAP and bus master design, MST_HWICAP were studied. DMA_HWICAP integrates a DMA controller with an interface to the HWICAP. The DMA controller has two interfaces, a slave interface to receive commands for starting a transfer, and a master interface to perform the transfer from memory to the ICAP. This design achieved an average reconfiguration speed of 82.1 MB/s. MST_HWICAP optimizes the DMA_HWICAP by using an integrated bus master with burst support. This removes the communication overhead between the DMA and HWICAP. This design achieved

an average reconfiguration speed of 234.5 MB/s.

The most trivial design is the XPS_HWICAP. This is the ICAP wrapper IP provided by Xilinx that attaches as a slave to the PLB. In the study, the average reconfiguration speed for this interface was 19.1 MB/s when transferring data from the PowerPC with cache enabled. The benefit to using this design is that Xilinx provides the hardware and a Linux device driver. This means that in theory, this design would be the easiest to get working. That is the approach taken in this thesis. The goal is to integrate partial dynamic reconfiguration. However, more advanced ICAP interfaces have been designed as described above. These types of implementations were beyond the scope of this thesis.

Chapter 4

Framework Overview

This chapter describes the developed framework for providing DMA transfers and partial dynamic reconfiguration in a commodity FPGA cluster. First, the design decisions leading to the proposed framework will be presented. Next, the high level cluster organization will be discussed. Then the operating system and software environment used will be discussed. From there, a high level description of the system hardware will be given. A detailed description of the developed hardware for managing hardware accelerator data transfers and configuration will be given. The developed Linux device driver will be discussed in detail to show how user requests are handled in kernel space and are then transferred to the developed hardware. Finally, the general process of deploying an application both in terms of hardware and software design is discussed.

4.1 Design Decisions

4.1.1 Data Transfer Methodology

One of the most straightforward ways of connecting user hardware logic to the PowerPC in Virtex5 FXT devices is through a FIFO interface on the Processor Local Bus (PLB). The Xilinx Platform Studio software has a graphical interface to make adding a FIFO peripheral very easy. In [6], a basic character device driver was developed to support communications with such devices from Linux. The driver uses the iowrite32_rep and ioread32_rep Linux functions to write and read 4 byte words to and from the target device. The iowrite32_rep and ioread32_rep functions use processor load and store instructions for reading and writing

to the appropriate address on the PLB. Since the PowerPC440 core only supports beat transfers for load and store instructions, these transfers will not take advantage of bursting that the PLB offers and will be limited in throughput. The reported maximum throughput achieved in the original work using the driver was 42 MBps for the same Virtex-5 FXT hardware being used in this thesis. This limited throughput serves as a major bottleneck for applications with high communication-to-computation ratios. In fact, more performance can be squeezed out of the driver by making a few modifications to the hardware and driver use scenario. In the original work, small read and write fifos with sizes of only 4 words were used. In addition, the application performed separate small reads and writes frequently, rather than a small number of larger reads and writes. To determine the best possible throughput the PLB FIFO architecture could achieve, a modified system was designed. In this system, large read and write FIFOs of 1024 words were used with a simple loopback hardware design that forwards data in the write FIFO to the read FIFO. Singe read and write operations varying in size from 4 words to 1024 words were performed. Figures 4.1 and 4.2 show ChipScope captures of 4 word write and read operations on the PLB bus.

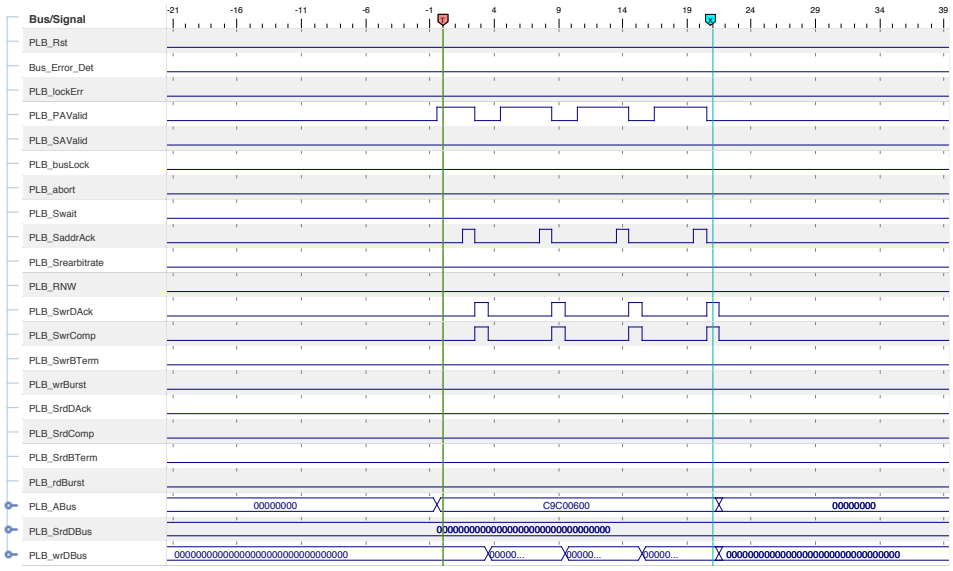

Figure 4.1: PLB Write Four Words from PowerPC

As show by the cursors in the figure, it takes 21 cycles to complete the write operation

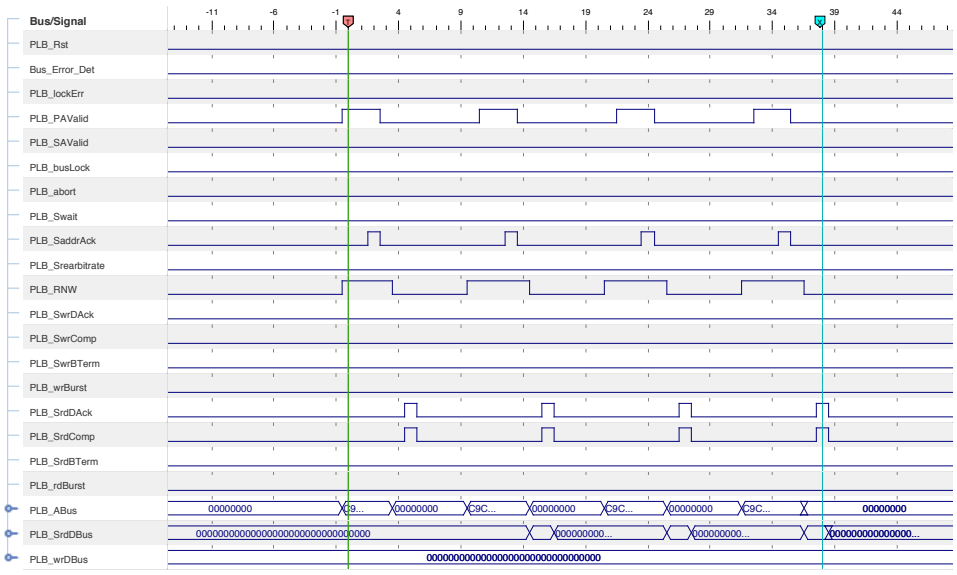

Figure 4.2: PLB Read Four Words from PowerPC

of four words. With a PLB clock of 100MHz, this equates to a throughput of 72.66 MBps write transfer speed. This is the maximum achievable speed as the overhead cycles between data beats for large transfer sizes cause the throughput to linearly drop off. Although this is what is achievable on the PLB with the system configuration, it does not take software overheads into account. Therefore, even in the best case scenario, throughput in this design is limited and alternative approaches are necessary to improve performance further.

Since the Virtex II Pro, the Virtex-4 and Virtex-5 series FPGAs have come to market. The Virtex-5 FXT architecture is the first in the series to add silicon scatter gather DMA controllers. These DMA controllers have yet to be used in any documented general purpose accelerator framework, but have a lot to offer. Xilinx has two application notes for designing a LocalLink peripheral and using it from Linux [19][20]. In a Xilinx test performed on a ML-507 board based on a Virtex-5 FXT device, a throughput of 476.1 MBps was achieved with the HDMA controllers compared to 178.7 MBps for the Central DMA controller [18]. In this test, a packet size of 2 KB, and total transfer of 260 KB was used for the HDMA controller. The reported bandwidth measurements do not include operating system overheads. For HDMA transfers, the throughput was measured from the start of the

first LocalLink frame to the end of the last frame in hardware.

Not only do the HDMA controllers offer more than twice the throughput of of the Central DMA Controller, but there is also potential to save FPGA resources. The Central DMA controller in the minimal configuration for a Virtex-5 device requires 280 slices, 481 flip flops, and 687 LUTs [22]. Xilinx also has a soft core scatter gather DMA controller which could potentially offer higher performance than the Central DMA controller, however it is not supported on Virtex-5 devices [23]. The HDMA LocalLink interface offers the highest bandwidth connection with the system DDR2 memory while consuming relatively little FPGA resources and therefore is the best choice for high throughput accelerators. One issue is that there are only four HDMA controllers per processor block. One is already used in the system design by the Hardware Ethernet MAC. This means there are three dedicated DMA channels left for used by custom hardware accelerators. If more than three accelerators are to be used, a single DMA channel will need to be shared amongst multiple accelerators. Thus, the framework will implement the capability to perform data transfers from the PowerPC using an HDMA controller that is shared between multiple accelerators.

4.1.2 Partial Reconfiguration Methodology

The goal of this framework is to support partial dynamic reconfiguration integrated with the developed framework. As such, an emphasis was not placed on performance of reconfiguration. The most straightforward approach to integrate partial dynamic reconfiguration was to use the Xilinx HWICAP IP and modify the Linux device driver to work with the Virtex-5 FXT. There are better approaches that could be used to achieve faster throughput from the CPU to the configuration memory, however these approaches are more complex custom solutions and do not fit within the scope of this work.

4.2 Cluster Configuration

A network diagram of the cluster configuration is shown in Figure 4.3. The cluster is organized in a beowulf configuration. Users from an external network connect to the head node of the cluster, fpga.ce.rit.edu. The head node is a 3GHz Core2 Duo system with 8GB Memory. It is running CentOS 5.5 Linux, and has the Xilinx Design Suite 12.1, ELDK cross compile tools and Modelsim installed. Once connected to the head node, the compute node portion of the cluster can be accessed. The compute nodes are connected to the head node with a 16 port Gigabit ethernet switch. Each compute node is a Virtex-5 FXT based Xilinx ML-510 development board running Angstrom Linux with a custom kernel supporting FPGA hardware features. Each ML-510 board has 512MB of DDR2 Memory, gigabit ethernet hardware, and boots from a compact flash card. OpenMPI 1.4.1 is installed to facilitate message passing between the compute nodes.

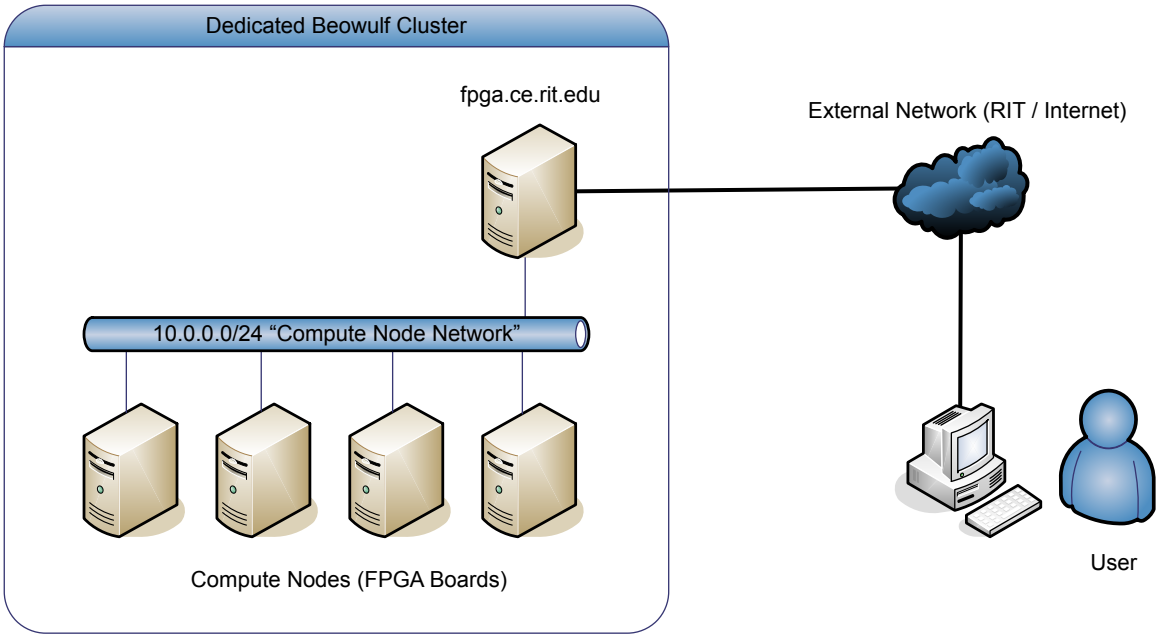

Figure 4.3: Beowulf Cluster Logical Network Diagram

4.3 Software Environment

The first step in the high level design of the framework was to setup an operating system to run on the embedded PowerPC processor. There are multiple options for embedded operating systems on the PowerPC processor, but Linux was chosen primarily due to the availability of device drivers for Xilinx IP, previous research, and personal preference. There are two primary components that make up an embedded Linux installation, the kernel and the root file system. Each of these components will be discussed in the two following sections.

4.3.1 Angstrom Distribution

There are various embedded distributions of Linux, both from commercial vendors and the Linux open source community. In order to make the framework more accessible, a constraint was set to use a community distribution rather than a commercial offering. One popular Linux distribution for embedded systems is Angstrom. Angstrom offers a package management system similar to desktop implementations. This makes creating, installing, and distributing software packages more convenient. For these reasons, Angstrom was chosen to generate the root files system for the system.

4.3.2 Linux Kernel

The Linux kernel used in this work is xilinx_v2.6.33 with added device drivers and other miscellaneous customizations. The kernel tree for this work is kept in a git repository which can be accessed online at http://fpga.ce.rit.edu. The kernel contains drivers for Xilinx specific IP such as the XPS_LL_TEMAC ethernet controller. Additionally, drivers for PLB slave FIFO accelerators and the framework developed in this thesis are included.

4.4 System Hardware Design

Figure 4.4 shows a high level diagram of the hardware system architecture created in this framework.

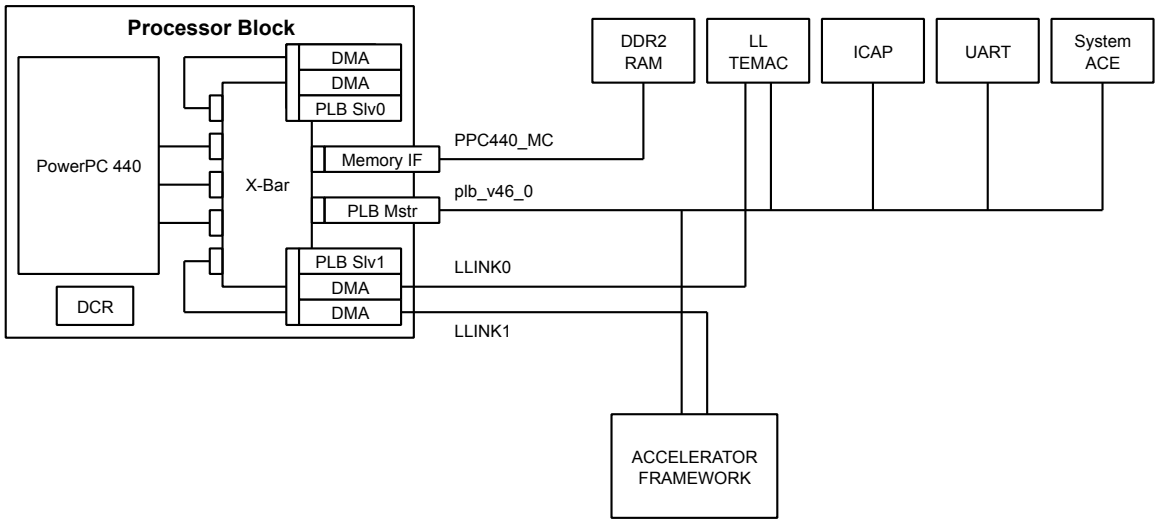

Figure 4.4: System Hardware Architecture

The system contains standard peripherals including a UART for serial output and debugging, System ACE controller for booting from Compact Flash cards, LL_TEMAC for ethernet communications, and DDR2 memory. Additionally, the framework uses the HWICAP peripheral developed by Xilinx and accelerator framework hardware developed as part of this thesis. The accelerator framework utilizes a LocalLink connection from a hardware DMA controller to facilitate DMA transfers between the CPU and hardware accelerators. It also has a PLB interface for registers to control the framework and read back debug information. LocalLink is a synchronous serial protocol developed by Xilinx and used for communication with some of their IP. Figure 4.5 shows an example of the Local Link protocol.

There are a total of eight signals and a data line. Data is transmitted over the interface in packets. Transmits leaving the DMA controller to hardware have an eight byte header, followed by the actual data. The header data is used by the accelerator framework hardware to route packets to the appropriate accelerator.

Figure 4.6 shows a block diagram design of the accelerator framework.

There are two interfaces to the hardware. There is the full duplex LocalLink connection which is why TX and RX are shown separately. There is also a PLB slave interface which

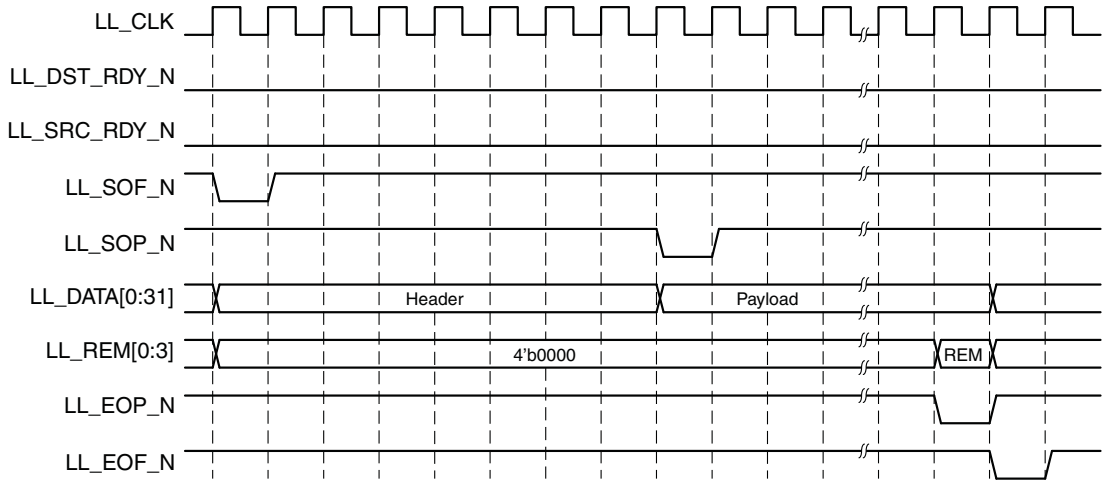

Figure 4.5: LocalLink

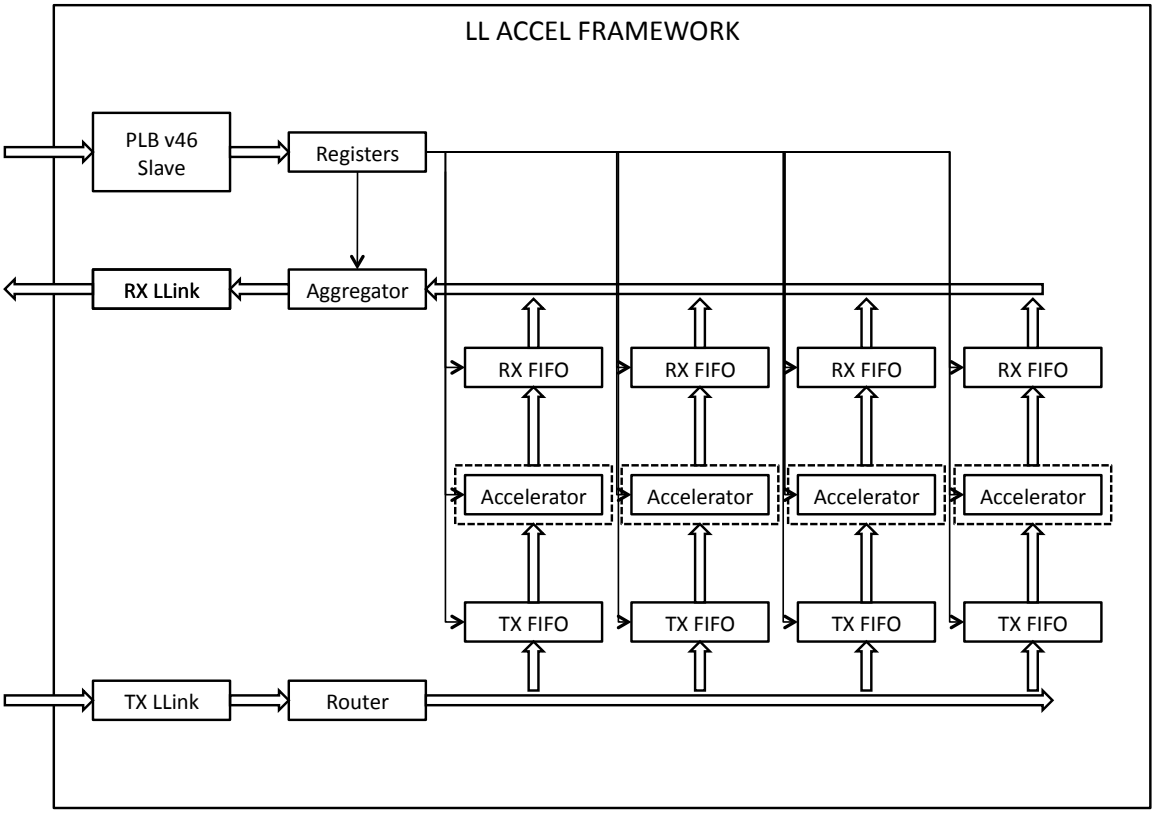

Figure 4.6: Framework Hardware

has registers that are readable and writable from software. These registers are used for control of the framework during data transfers and reconfiguration. Four accelerators were

chosen to share a DMA channel in this work, which is represented in the diagram. Each accelerator has a TX FIFO and RX FIFO. The FIFO sizes used in this thesis were chosen to be 8KB to allow sufficient experimentation. The accelerator regions have a dashed box around them to indicate the reconfiguration interface. After an accelerator is reconfigured, the accelerator and FIFOs are pulled to reset through a control register. This is to prevent reset the accelerator to it's default state.

Figure 4.7 shows a block diagram of the accelerator interface.

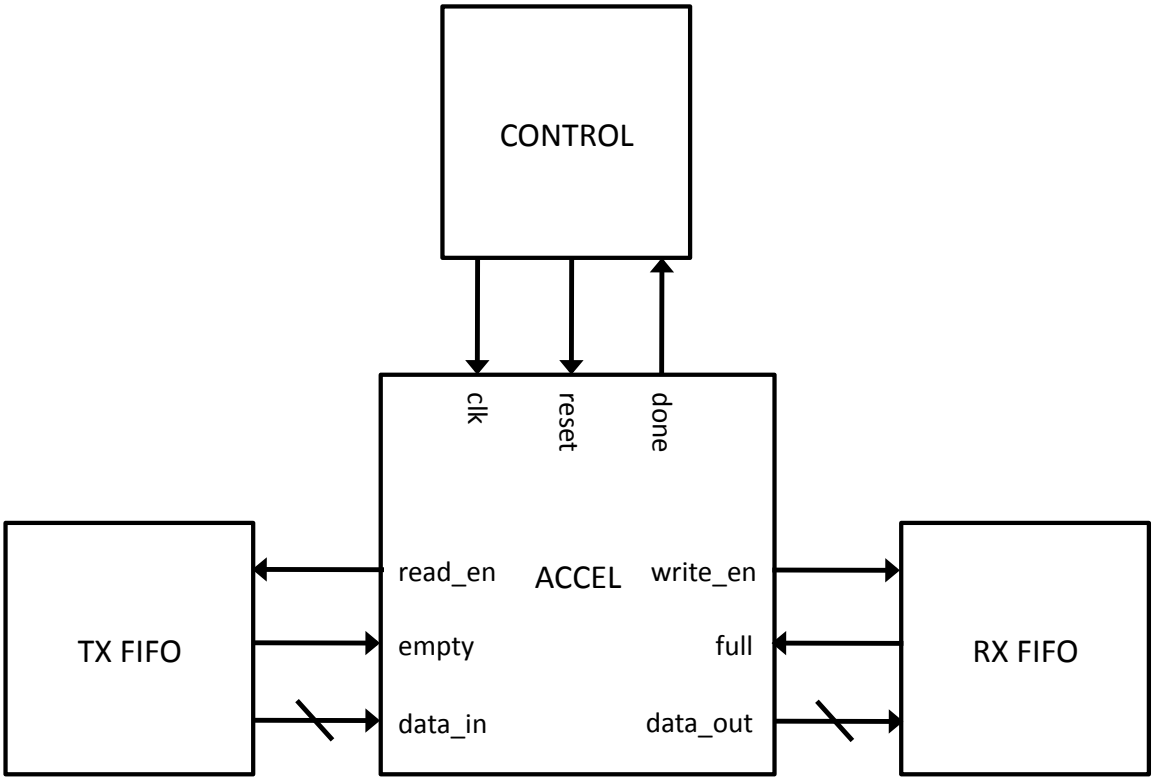

Figure 4.7: Accelerator Interface Signals

Hardware designers have a simplified interface of only nine signals. There is read_en to request data from the Tx FIFO, empty to indicate the Tx FIFO is empty, and data_in which is the data line from the Tx FIFO. To write out results from the accelerator, there is a write_en signal to the Rx FIFO, full signal to indicate that the Rx FIFO is full, and data_out data line for data to the Rx FIFO. There are clk and reset inputs from the framework. A

done output is asserted by the accelerator when it has finished processing data. Accelerators developed to target the framework will require a minimal state machine implementation to read data out of the Tx FIFO and write results into the Rx FIFO.

Since four accelerators were chosen for this work, four reconfigurable partitions had to be laid out in the FPGA floorplan. Figure 4.8 shows the layout of the reconfigurable partitions.

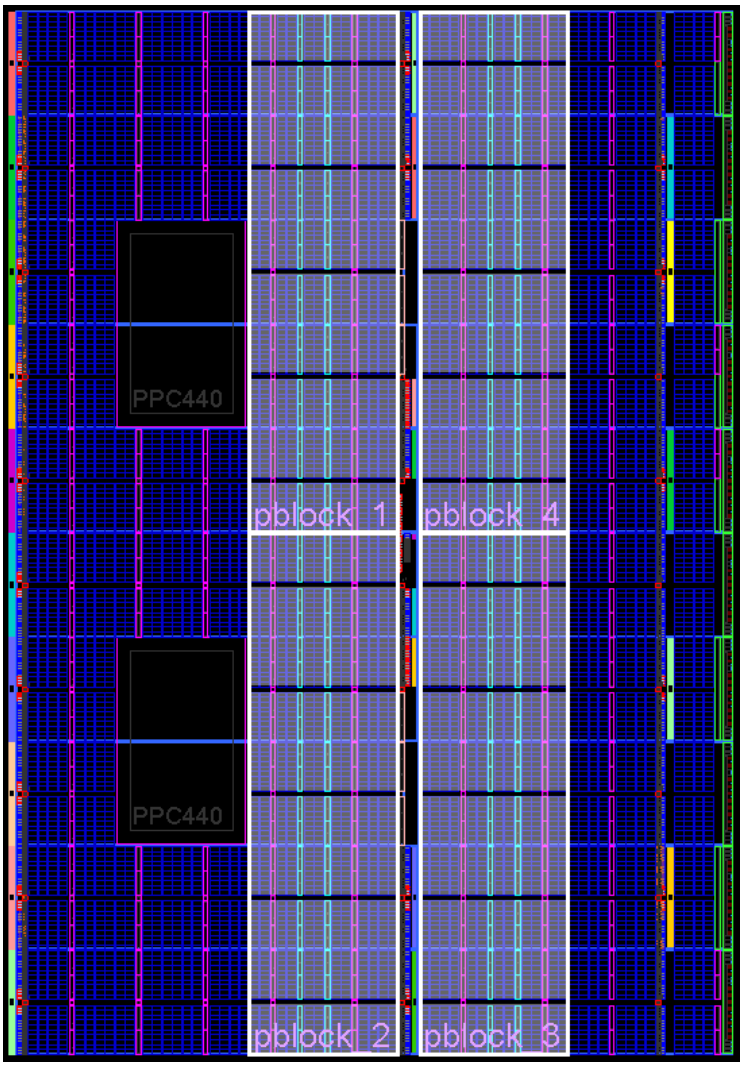

Figure 4.8: FPGA Partitioning Floorplan

Each partition is sized to contain an identical number of constrained resources. The

resources reserved by each partition are listed in Table 4.1.

Resource	Available
LUT	9600
FD_LD	9600
SLICEL	1500
SLICEM	900
DSP48E	80
RAMBFIFO36	40

Table 4.1: Partition Resources

4.4.1 Device Tree Specification

The PowerPC system configuration is written to a dts file. There is no concept of a bios that holds the system configuration settings for the PowerPC architecture. All of the system hardware must be specified in the dts file. The dts file contents are read in by the kernel at boot time to know what hardware exists on the system. A dts file can be generated from an XPS project. Figure 4.9 shows the entries in the dts file necessary to use the framework hardware. It is also assumed that DMA1 channel has been enabled in XPS.

```
ll_accel_read_control_0: ll-accel-read-control@c2a00000 {
        compatible = "xlnx,ll-accel-read-control-1.00.a";
        interrupt-parent = <&xps_intc_0>;
        interrupts = < 1 2 >;
        reg = < 0xc2a00000 0x10000 >;
        llink-connected = <&DMA1>;
        xlnx,family = "virtex5";
        xlnx,include-dphase-timer = <0x1>;
} ;

xps_hwicap_0: xps-hwicap@86800000 {
        compatible = "xlnx,xps-hwicap-4.00.a", "xlnx,xps-hwicap-1.00.a";
        reg = < 0x86800000 0x10000 >;
        xlnx,bram-srl-fifo-type = <0x1>;
        xlnx,family = "virtex5";
        xlnx,read-fifo-depth = <0x80>;
        xlnx,simulation = <0x2>;
        xlnx,write-fifo-depth = <0x40>;
} ;
```

Figure 4.9: Dts File Entries

4.5 Linux Device Driver

A character device driver was developed to take requests from user space and control the developed hardware to perform the specified task. The are different ways that communication with hardware from the Linux kernel could be handled. A custom system call could be implemented. The mmap functionality could be used to map hardware to used space, and other various options. Ultimately a standard character device driver was chosen. The reason for this is it is a well established method to communicate with hardware that is stream based. Figure 4.10 shows a diagram representing the general flow from a user space program all the way down to hardware and back. This section will focus on describing the kernel layer of the diagram and how it interacts with both user space and hardware.

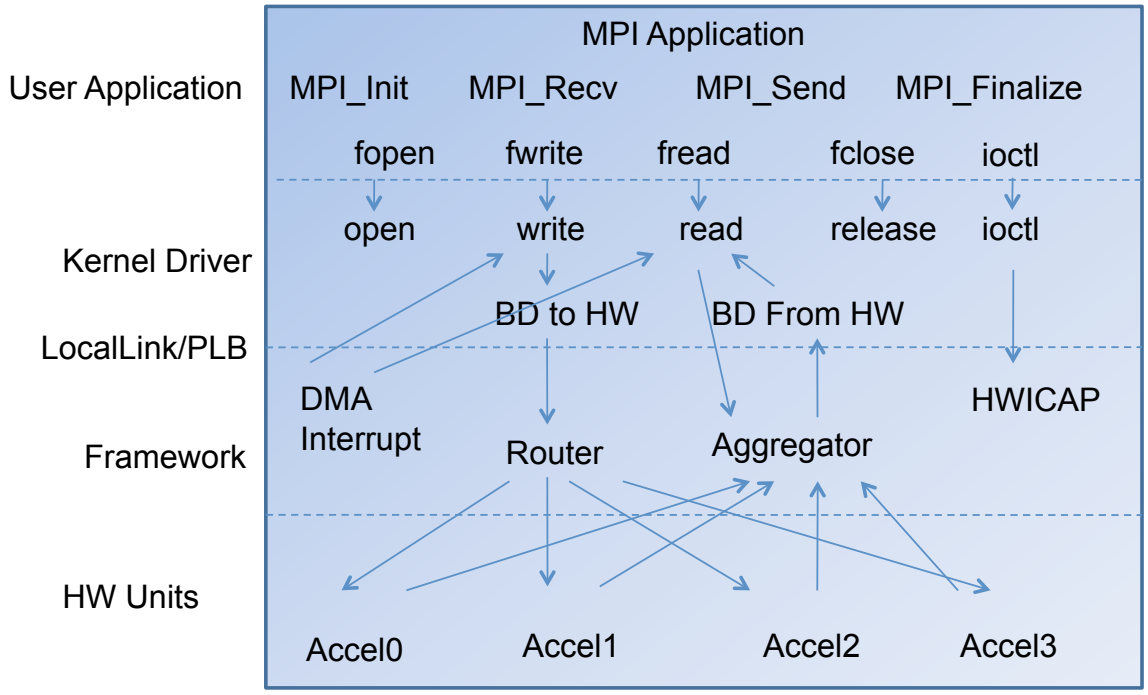

Figure 4.10: User Space to Hardware Diagram

The system calls shown in Table 4.2 are implemented by the device driver. More details of each call will be discussed as implementation details of the driver are shown.

System Function Call	Usage
open	Claim accelerator
release	Free accelerator
write	Transfer data via DMA operations
read	Receive data via DMA operations
ioctl	Reconfigure accelerator

Table 4.2: File Operation Mappings

4.5.1 Driver Structure

A single instance of the driver is used to control all hardware accelerators. The driver needs to be given the number of accelerators, which can be done in multiple ways. One way would be to pass it as an argument if starting the driver with insmod. Another possibility would be to include a field in the dts file that could be parsed out by the driver when it is started. In this work, a default value of 4 accelerators assumed, but this could be easily changed in the future. Each accelerator is given a unique minor number, and an associated data structure is kept for each accelerator in the driver. The driver creates four devices in /dev upon successful startup. Figure 4.11 shows a listing of /dev and the hardware accelerators created by the driver.

```
# ls -la /dev
...
crw-------    1 root     root      252,   0 Jan  1  1970 ll_accel0
crw-------    1 root     root      252,   1 Jan  1  1970 ll_accel1
crw-------    1 root     root      252,   2 Jan  1  1970 ll_accel2
crw-------    1 root     root      252,   3 Jan  1  1970 ll_accel3
...
```

Figure 4.11: Directory Listing of /dev

Each device has the same major number of 252 which is dynamically assigned by the kernel and may vary depending on the number of other major numbers consumed by other devices. The minor numbers 0 through 3 can be seen for each accelerator as well. The driver looks for two entries in the kernel dts file. One is for ll_accel and the other is for hwicap. The reason for this is that the HWICAP IP can be added in XPS and the driver can

find it without making any manual modifications to the dts file such as copying the hwicap information into the ll_accel device declaration.

4.5.2 DMA Operations

The read and write system calls transfer data between the processor and hardware buffers using DMA operations. Xilinx provides Linux code (lldma) to manage DMA descriptor rings as well as an example driver [20] which this work used as a starting point. Each DMA controller in the PowerPC processor block is controlled with Device Control Registers (DCR) from the CPU. This developed framework allows multiple accelerators to share a DMA channel. This capability is provided through mutual exclusion in the driver. When an accelerator performs a write operation, it locks a mutex from the point it setup the Buffer Descriptor to the point when a TX interrupt comes back. When an accelerator performs a read, a mutex is locked from the time the read request is sent to the time an rx interrupt comes back and the data is precessed.

When a write call is performed, the driver copies the data buffer specified from user space to a kernel buffer. A buffer descriptor is then set up to tell the DMA controller what data to transfer. The cache is invalidated for this memory and the buffer descriptor is enqueued to hardware for processing by the DMA controller. Figure 4.12 shows the translation of a buffer descriptor to a LocalLink packet going out to hardware.

When a read call is performed, the driver requests that the hardware sent up results by writing to a slave register over the PLB. The driver then blocks until a RX interrupt from the DMA controller is received. Once the interrupt is received, the Buffer Descriptor is processed and the data is copied from kernel space to a user space buffer.

4.5.3 HWICAP Partial Reconfiguration

Xilinx provides a driver for the HWICAP, however the framework needs to control reconfiguration of all accelerators. It would not be good to allow anyone to access the HWICAP

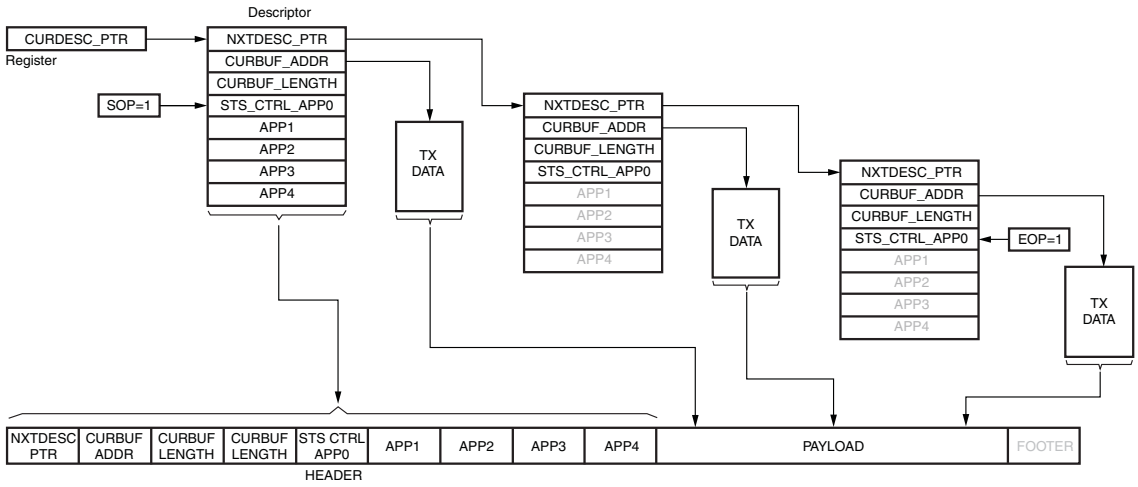

Figure 4.12: Assembly of Tx Data Packet

separately and overwrite a users hardware accelerator. Instead, the hwicap driver was integrated into this device driver. Some modifications were made to enable the driver to work with the Virtex-5. The main modification required was to switch the writing method to a per word write rather than a buffering approach. This reduces potential throughput but was necessary to get the hardware working correctly. An ioctl call is made by the user passing in a binary partial bitstream and the size of the bitstream. The driver then writes the bitstream to configuration memory, reconfiguring the target accelerator. Currently, there is no protection against a user sending a bitstream that could overwrite a different accelerator, however this protection could be implemented in the driver by examining the configuration data. A better solution would be to use dynamic bitstream relocation in which a single partial bitstream could be dynamically applied to any of the reconfigurable partitions.

The ioctl call takes pointer to a buffer. When the ioctl system call is called from user space the following actions occur. The bitstream data is copied from user space to kernel space. The data is then written over the PLB to the HWICAP peripheral. The HWICAP writes the bitstream data to FPGA configuration memory. After all of the bitstream data has been written, a reset signal is asserted to the accelerator that has just been reconfigured. This puts the accelerator into it's default state.

4.6 Programming Model

A simple example program was written to demonstrate how to use the developed framework to reconfigure an accelerator from user space. An example usage of the program usage is as follows: configaccel /dev/ll_accel0 partial_bitstream.bin. The first parameter is /dev entry of the accelerator to reconfigure. The second parameter is the partial bitstream to use. The partial bitstream has the bin extension because header information has been stripped off by using adding a -g Binary:Yes parameter in bitgen generation. Figure 4.13 shows the reconfiguration program.

The following analysis of the program demonstrates how to perform reconfiguration from user space. The first matter of importance is the command number. This is represented in the LL_ACCEL_IOCSBITSTREAM define. This number should be unique to the device to prevent issues with the same command being used across multiple devices. The command codes are composed of several bitfields. A portion of the number is ”magic”, which is unique to the device, while additional command will sequentially increase from the magic portion. There is not specific meaning to the value of the command other than it is the command used in this device driver to reconfigure the target accelerator. Next is the bitstream_t struct definition. The device driver expects a user space buffer in the format with an unsigned int specifying the size of the data followed by a pointer to the actual bitstream data. In main, the accelerator device file is opened. Next the bitstream file is opened. The bitstream is read into a buffer and the size of the file is saved. The bitstream_t struct is setup to reflect the file data. Finally, the ioctl system call is made. The ioctl call specifies the device being targeted, the command to execute, and a parameter to the command which contains the bitstream data.

Data transfers to the accelerators are performed by using the write and read system calls. These are standard calls and don’t require any further explanation. The only advice would be to try and partition data transfers so that the data is transmitted in the largest chunks possible. This will ensure that better throughput through the DMA controller is attained.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>

#define LL_ACCEL_IOCSBITSTREAM 0x80047801

typedef struct
{
    unsigned int size;
    unsigned char *data;
} bitstream_t;

int main(int argc, char *argv[]) {
    int accel;
    int i,ret, fileSize;
    char *buffer;
    FILE *bitstream;
    bitstream_t ioctlInfo;

    accel = open(argv[1], O_RDWR);
    if (accel < 0 ) {
        printf("Error opening File: %d\n", accel);
        printf("ERRNO: %d\n", strerror(errno));
        exit(1);
    }

    bitstream = fopen(argv[2], "r");
    if (bitstream == NULL) {
        printf("File Not Found\n");
        exit(2);
    }

    fseek (bitstream, 0 , SEEK_END);
    fileSize = ftell (bitstream);
    rewind (bitstream);

    buffer = (char*) malloc (sizeof(char)*fileSize);
    if (buffer == NULL) {
        printf("Memory error\n");
        exit(3);
    }

    ret = fread (buffer,1,fileSize,bitstream);
    if (ret != fileSize) {
        printf("Reading error\n");
        exit(4);
    }

    ioctlInfo.size = fileSize;
    ioctlInfo.data = buffer;

    ret = ioctl(accel, LL_ACCEL_IOCSBITSTREAM, &ioctlInfo);
    if (ret < 0) {
        printf("ERRNO: %s\n", strerror(errno));
    }

    close(accel);
    free(buffer);
    fclose(bitstream);
}
```

Figure 4.13: Reconfiguration Example Program

4.7 Hardware Accelerator Creation and Deployment

The default framework provides four accelerator slots. Ideally, applications will target this configuration as it will require the least amount of work to get something working. In the event that a different partitioning scheme is desired, it can be changed but will require rebuilding the static firmware and retargeting all reconfigurable partitions for this design. In this overview, it will be assumed that an application is being deployed to the established partition layout.

The first step is to write the HDL for the accelerator being deployed. Each accelerator has a common interface consisting of nine signals. The general procedure is to read data out of the Tx FIFO, perform processing, write to the Rx FIFO and assert the done signal when all expected data has been processed. After the HDL has been written, the accelerator design is verified using the developed framework testbench in Modelsim. The testbench simulates from LocalLink data coming in on the Tx interface, through the Tx FIFO to the accelerator, out of the Rx FIFO and out over the Rx LocalLink interface. This allows the designer to be confident that their design will work in hardware once simulation is successful. The next step is to synthesize the accelerator. To do this, an ISE project is created for the target FPGA. The HDL source code is imported into the project. The synthesis properties are modified to disable adding I/O Buffers. After synthesis is complete, a netlist is generated for the accelerator. The netlist is then added to the PlanAhead project. If the accelerator is to target any of the four partitions, it will need to be implemented for each one. After the accelerators have been implemented, partial bitstreams need to be generated. The parameter -g Binary:Yes must be used in bitgen to strip off the header from the bitstream. This is because the driver is expecting raw configuration data, so there should not be any meta data in the file. Once the partial bistreams have been produced, they can be copied to the FPGA boards over the network. From there software applications can reconfigure accelerators by using the ioctl system call for the specified device in any user space application. A loopback accelerator was created for testing DMA throughput and

partial reconfiguration capabilities. The following section details the design to give a more concrete example of framework application deployment.

4.7.1 Loopback Accelerator

The focus of this work is on an improved framework for FPGA cluster computing and not any particular computational application. As a result, a simple loopback accelerator was the only hardware necessary to validate design goals. A loopback device forwards incoming data directly to the output memory without performing any computations on the data. Accelerators in the framework must read from an input fifo, perform the necessary computations, write to an output fifo, and assert a done signal to notify the framework that data is ready to be read. This accelerator was used in determining some of the performance results in Chapter 5.

4.7.2 Hardware

The loopback accelerator hardware is a straightforward state machine. Figure 4.14 shows a diagram. The hardware will wait in the idle state until data has been written to the accelerators input fifo. Once the fifo has data, the accelerator reads the first word which is the control word. In this example, only one control word is necessary, as it specifies the number of following words that are to be transferred. The accelerator saves this number to a 32-bit register for keeping track of when it has processed all of the requested data. While the input fifo is not empty, the accelerator keeps reading words and writing them to the output fifo until the specified number of words has been read. Once this condition occurs, the accelerator goes to the done state where a signal is asserted to the framework indicating data is ready yo be read out of the accelerators output buffer to be sent back to the CPU. The accelerator does not perform any check to see if the output buffer is full because the by design, the accelerator can process at most as much data as the output buffer can hold before signaling for the data to be sent to up to the CPU.

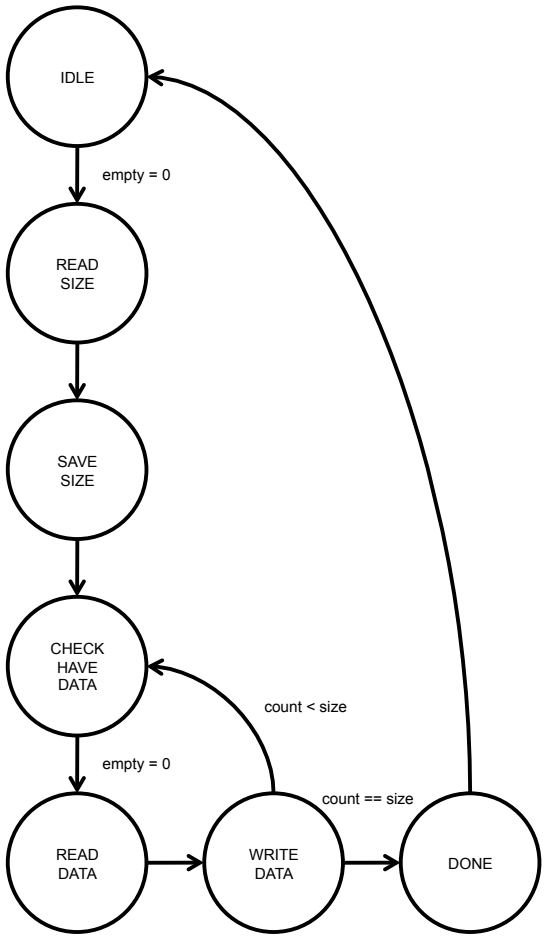

Figure 4.14: Loopback Accelerator State Machine

4.7.3 Software

The software to send and receive data to the loopback accelerator is a straight forward application of the framework. Each transfer to the accelerator is performed with an fwrite operation. Since control for the accelerator is embedded in the data stream, the first word contains the size in words of the transfer to be performed. It is the software developers responsibility to provide a corresponding fread to read back the data sent back from the accelerator before the fifo overflows. Transfers are done in this way to maximize the packet size and therefore get the best throughput possible.

Chapter 5

Performance Analysis

5.1 Methodology

Various timing mechanisms were used to benchmark performance of the developed framework. Single accelerator data transfer timing was performed in kernel space within the device driver using the PowerPC440 64bit timebase register. The timebase register is composed of two 32 bit registers. The lower register increments once every clock cycle, which in this system configuration is at a rate of 400MHz for a timing resolution of 2.5 ns. The upper register increments every time the lower register overflows and is not utilized for benchmarking purposes. All of the timebase benchmarking can be turned off with a compiler define. The timebase measurements are saved in the device driver and can be displayed through the drive /proc filesystem entry. When measuring the performance of multiple accelerators, timebase measurements were disabled. From user space, the Linux gettimeofday() system call was used. This call has a resolution on the order of a microsecond and provides the most accurate time measurements available from user space.

5.2 Gigabit Ethernet Performance

Although the ML-510 is a commodity development board, getting the gigabit ethernet controller working properly in Linux took quite a bit of work. In XPS, custom hardware modifications were made to the default design generated by BSB. RGMII functionality was enabled in the XPS_LL_TEMAC IP. This IP supports LocalLink DMA transfers to and from the CPU to improve throughput. Some settings were tweaked in an attempt to gain

maximum performance. The TX and RX FIFOs sizes were changed from 4KB to 32KB. On the Linux side, the MARVELL 88E1111 using GMII driver should be compiled into the kernel. With the Gigabit ethernet hardware and driver setup, the performance on a system level could be tested. To evaluate performance, the NetPIPE tool was used. NETPIPE performs ping-pong type tests where messages increasing in size are bounced between two processes. These processes can be over a network or on the same system. For the gigabit ethernet performance analysis, two different runs of the tool were made for two different network configurations. In the first network configuration, a standard Maximum Transmission Unit (MTU) size of 1500 Bytes was used. The MTU specifies the largest payload that can be transferred in an ethernet frame. A larger MTU means that less transfers can be performed for a given amount of data resulting in higher throughput due to lower processing overheads. For MTU set at 1500, tests were run for raw TCP throughput and MPI throughput between two nodes on the network. Then the MTU was set to the maximum size of 8182 Bytes supported by the XPS_LL_TEMAC. The same two tests were run again. Figure 5.1 shows a graph of the results.

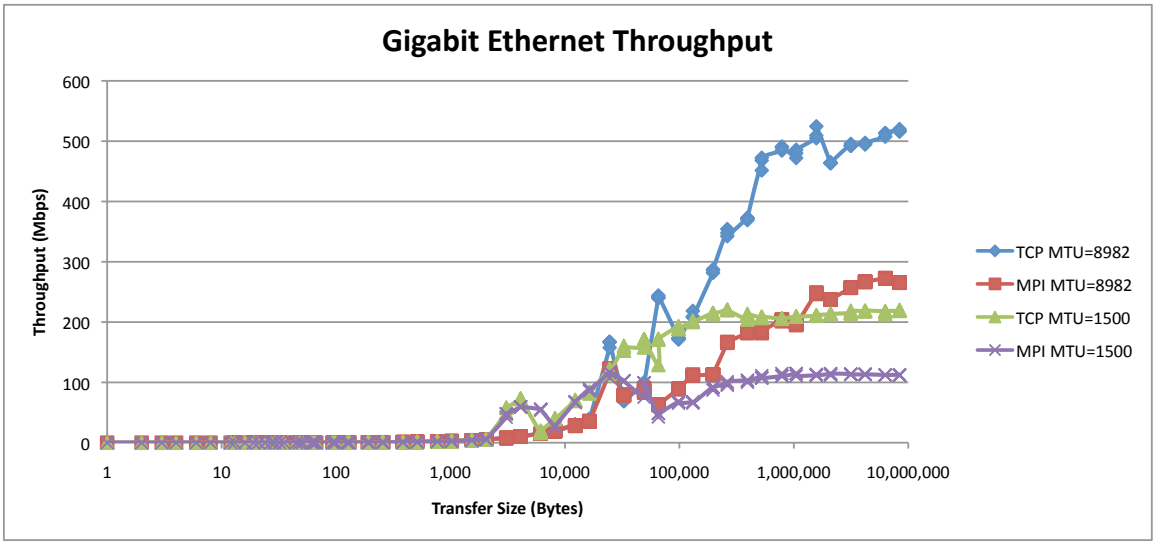

Figure 5.1: Gigabit Ethernet Throughput

It is clear that the MPI layer is adding significant overhead when compared to the raw

TCP layer it is utilizing. In the best case, raw TCP with a MTU of 8982 Bytes peaks at over 500 Mbps. For the intended use of the cluster however, the MPI times are the most important when analyzing performance. From the results, a smaller MTU will result in higher throughput up to about 100,000 Bytes. After that point, a larger MTU will start to yield better throughput. Depending on how much data will be transferred over the network in a deployed application, there could be a significant advantage in tweaking the network settings to suite the application.

5.3 MPI Single Node Process Throughput

Another important metric is the throughput between processes on the PowerPC440 using MPI. For this analysis, NetPIPE was used again. Figure 5.2 shows a graph of the results, which seem a bit strange.

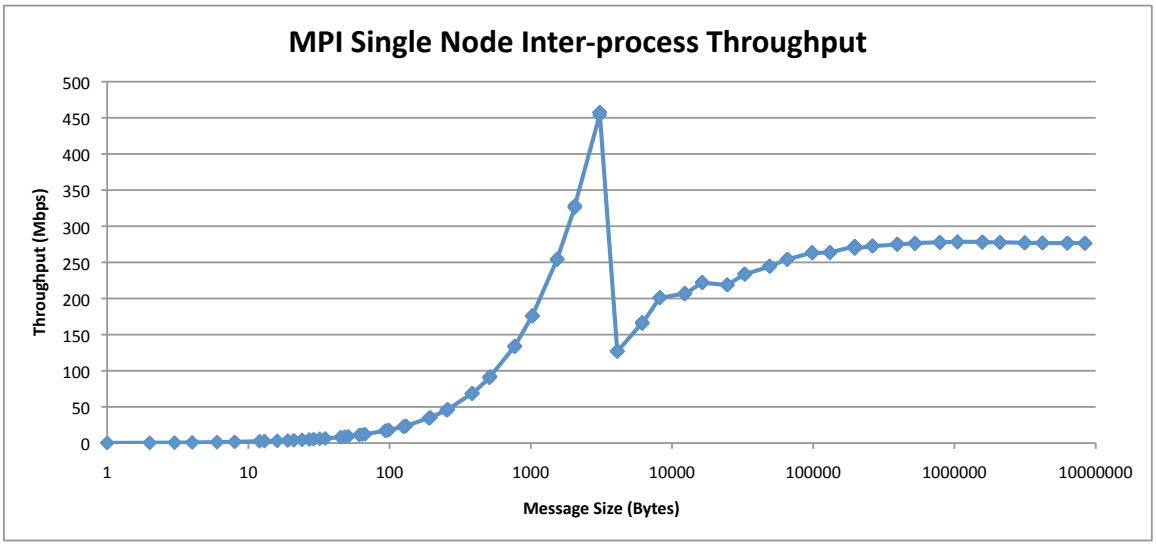

Figure 5.2: MPI Throughput

The throughput increases as you would expect up until around a 4KB message size. At this point the throughput dramatically drops off and then slowly starts to build and then settle at around 275 Mbps. This result appears to be a limitation of the memory management unit in the PowerPC440. The size of 4KB is significant because this is the

size of a page. It appears that once the size is surpassed, there is a significant performance hit that eventually settles out. This gives application developers an idea of what they could expect in terms of throughput between MPI processes on the same FPGA processor.

5.4 DMA Throughput Improvement

One of the primary design goals of this thesis was to improve the data throughput between the embedded CPU and hardware accelerators from previous works. The benchmark to compare against is a cluster framework supporting hardware accelerators connected to the PLB as slave devices. In terms of throughput, this approach is limited since transfers between PLB slaves and the CPU are limited to single beat transfers. In order to evaluate throughput performance, loopback accelerators were deployed targeting the framework developed in this thesis and a previously developed framework. Packet sizes ranging from 4 to 8192 Bytes were transferred between hardware accelerators in each framework. Figure 5.3 shows DMA throughput as measured in hardware.

The measurement is the mean of hardware TX and RX times measured over various packet sizes. For TX, the number of cycles from the TX SOF to the TX EOF were measured. For RX, the number of cycles between the RX SOF and RX EOF were measured. The results are in line with an application note from Xilinx [18] which reported a throughput of roughly 394 MBps. The reported throughput in the application note is higher because the LocalLink clock in that work was set at 133MHz, whereas in this work, the clock is set to 100MHz. With the DMA hardware measurements being as expected, the next step was to measure throughput from the kernel perspective. It is obvious that there will be some overhead in handing over data to the DMA controller and having it send it out to the hardware. From software, the throughput will be less, but how much less is the important factor since that will directly effect the bottom line of how fast transfers can be made. For the DMA measurements, on TX, time is started when a buffer descriptor is enqueued to DMA controller and stopped when a TX interrupt indicating the transfer has completed arrives

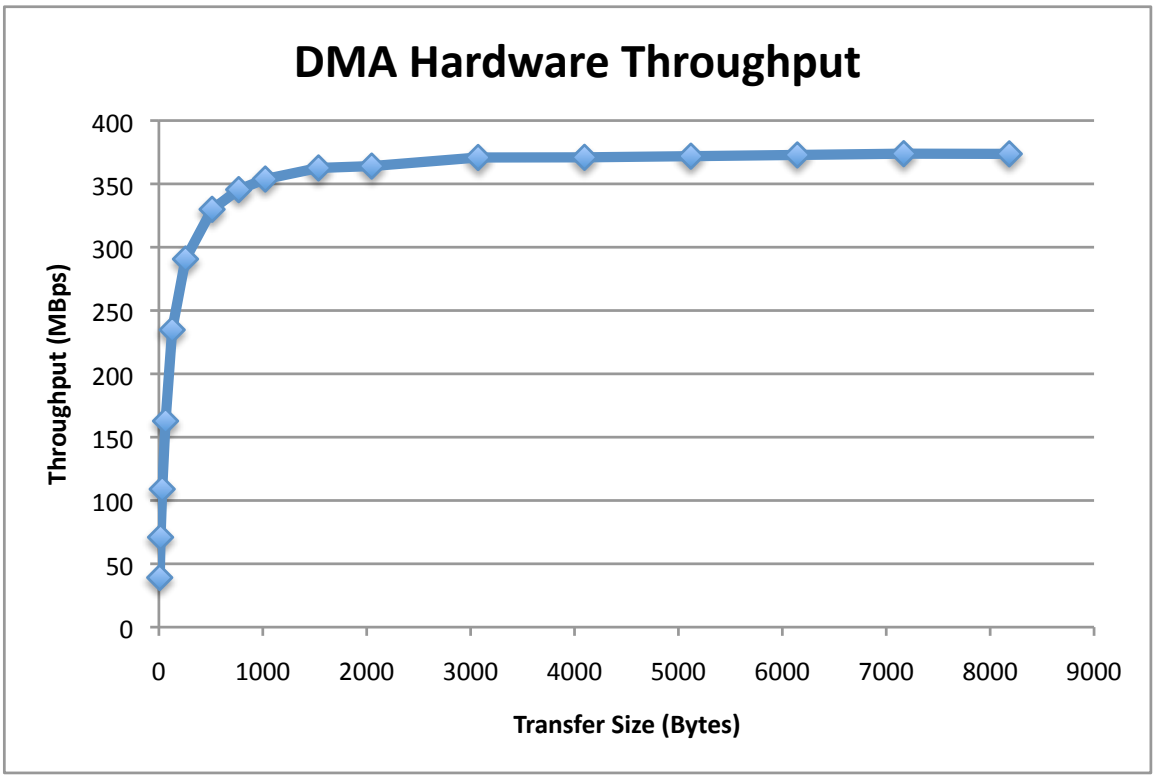

Figure 5.3: DMA Hardware Throughput

from the DMA controller. For RX, the time is started when the driver requests data from the framework through a PLB register write to the time an RX interrupt arrives indicating that the transfer has completed. Figure 5.4 shows a plot of the measured hardware through versus the measured throughput in the kernel. As shown, there is significant overhead for the call to complete in software and only as the packet size grows does it begin to approach the hardware throughput.

Figure 5.5 shows a graph of the PLB vs DMA. The results shown in the graph represent the average of the TX and RX throughputs measured for each framework. Throughput measurements are made in kernel space in the device driver. For the PLB, the measurements are made prior to and following the iowrite_32 and ioread_32 calls. For the maximum tested packet size of 8KB, the DMA transfers are 5.7x faster than the PLB transfers. For packets over 1KB in size, DMA transfers offer significantly increased throughput over a PLB slave

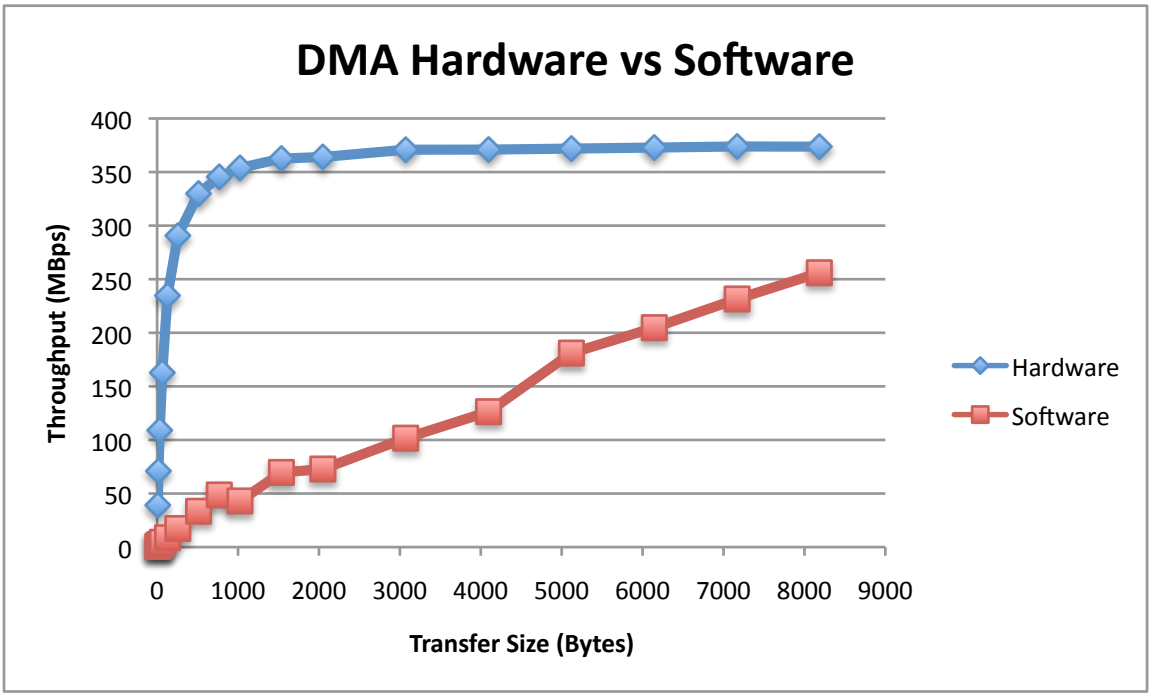

Figure 5.4: DMA Hardware vs Software

solution. This is an important characteristic to keep in mind when designing an application targeting this framework.

5.5 System Call Times

To further evaluate the performance of LocalLink DMA transfers against PLB slave transfers, the read and write system calls used in this study were analyzed to see how different phases of the system call were contributing to the overall time the call took to complete. In this analysis, a packet size of 8KB was used. Time measurements for each phase of the system call were made in the driver. Figure 5.6 shows a comparison of the system call breakdown for the DMA and PLB frameworks.

For the DMA calls, the overheads are significantly higher due to the need to manage buffer descriptors, invalidate cache, and other operations not necessary in the basic PLB driver. However, when the packet size is large enough, the actual transfer time is much

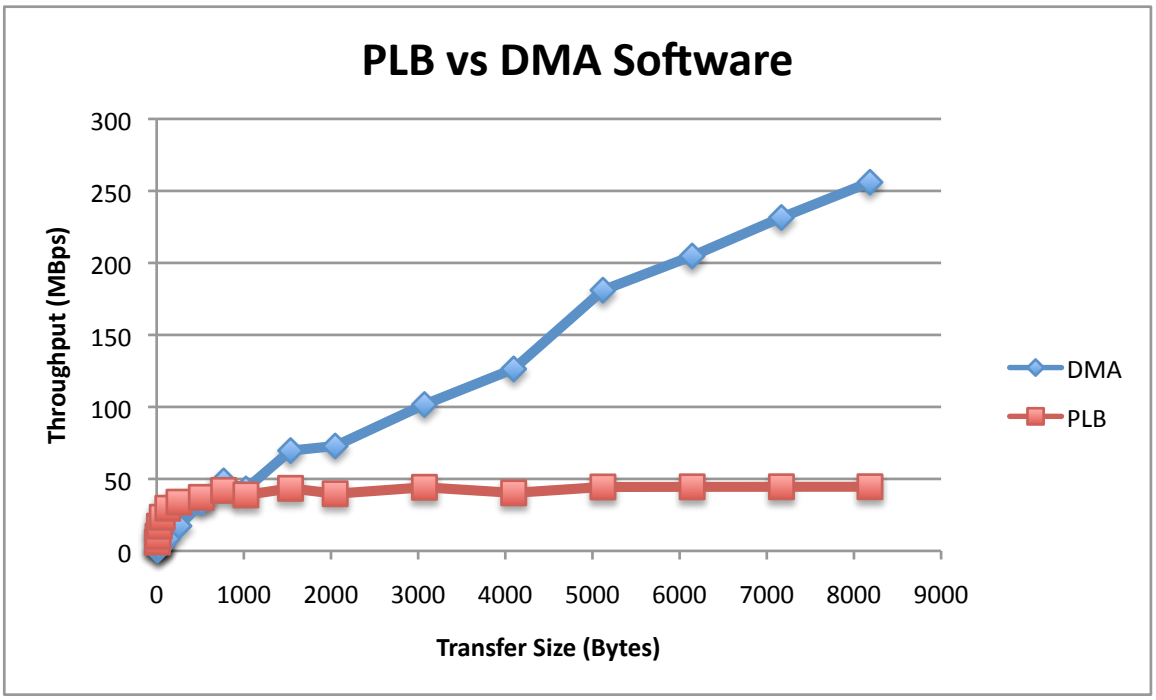

Figure 5.5: DMA vs PLB Throughput

faster outweighing the added overheads. It is clear in this case, DMA transfer times indicated by ACC2MEM and MEM2ACC are significantly lower as expected. The K2U portion which is copying data from kernel space to user space takes significantly longer for the DMA transfer than the PLB transfer. This is because this time has more overhead due to reading entries out of a list before performing the transfer. Read Accel Wait is the time the driver had to wait for the accelerator to finish processing. This time is broken out because, for throughput measurements, there should be no accelerator execution time included.

5.6 DMA Channel Scaling Characteristics

One notable feature of the developed framework is that a single DMA channel can be shared amongst multiple hardware accelerators. This arbitration is performed in the driver software through the use of mutual exclusion. To study how well that DMA channel is shared by an increasing number of hardware accelerators, the following tests were performed. An

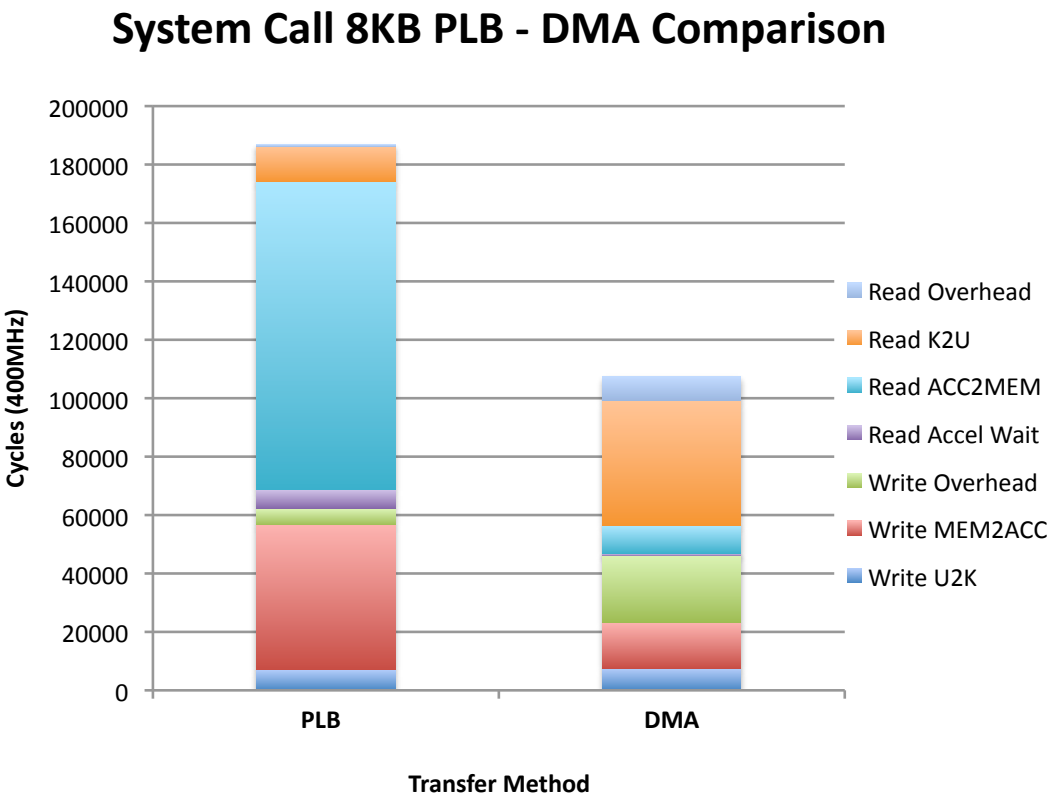

Figure 5.6: System Call Comparison

MPI program was written to transmit 10MB of data in 4KB packets between the CPU and hardware accelerators. The program was executed with a variable number of accelerators ranging from 1 to 4. A process was started to manage the transfers of each accelerator. The total execution time was measured following a barrier at startup to the time after a barrier when all processes had completed data transfers. This was performed for both the DMA and PLB frameworks for scaling comparison. Figure 5.7 shows the results.

The DMA has faster execution time than the PLB, however loses more ground each time a new accelerator is added. This is not too surprising considering the PLB is optimized for multiple transfers and has hardware arbitration. The DMA channel is arbitrated by a much slower software mutex and doesn't have more advanced features of the PLB. From these results, sharing multiple accelerators on a DMA channel won't scale well relative

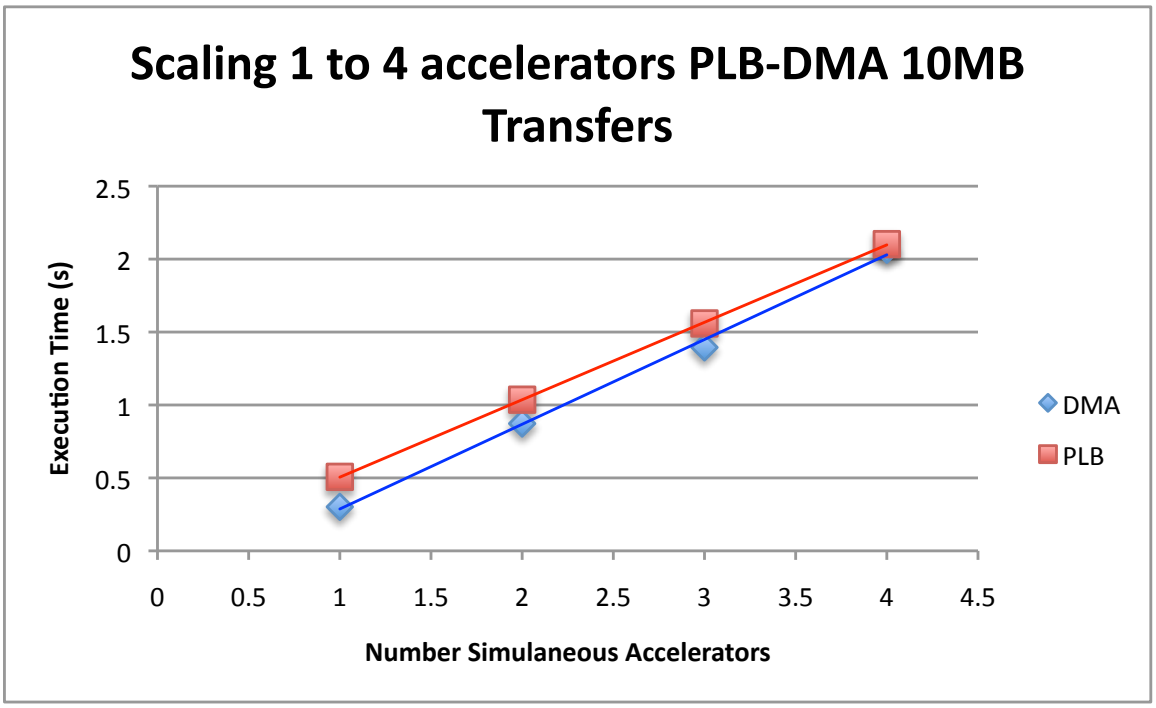

Figure 5.7: DMA Channel Scaling

to the PLB. However, if the number is limited and multiple DMA channels are used, it may make sense to have a larger number of accelerators in the framework. Additionally, if the accelerators are transmitting intermittently so as to effectively share the channel, the performance hit won't be as bad.

5.7 Reconfiguration Time

The floorplan used in this thesis partitioned part of the FPGA into four reconfigurable regions. These regions are defined by 661,668 Bytes of configuration memory. Performance measurements were made within the ioctl system call used to reconfigure a partition. It was determined that the configuration time was 136.2 ms for the partitions sizes chosen utilizing the HWICAP IP connected to the PLB. This results in a throughput of roughly 4.63 MBps. This is an important performance constraint especially for future applications considering taking advantage of dynamic reconfiguration as part of a computation cycle. Different sized

partitions would mean different amounts of time, but the study here gives a good middle of the road figure of what the configuration time will be using this framework. Research has been performed to study various methods to achieve the best throughput when performing reconfiguration. Table 5.8 shows the results of various designs studied in [31]. The design called XPS_HWICAP with the PowerPC cache enabled is the closest to the design used in the framework developed in this thesis. In the table, the average reconfiguration rate for this design is 19.1 MBps. This is greater than the 4.61 MBps measured in this work. There are a couple reasons for this. The first is that in this work, the measurements were made in the device driver kernel code and inherently include user space to kernel space copying as well as the actual data transfer over the PLB to the HWICAP. In [31], throughput measurements are made from the time the PLB master starts to feed data to the ICAP to the time the partial bitstream is completely downloaded into configuration memory.

ICAP design	**Test 1** (Bit. Size/Reconf. Time)	**Test 2** (Bit. Size/Reconf. Time)	**Test 3** (Bit. Size/Reconf. Time)	**Test 4** (Bit. Size/Reconf. Time)	**Avg. reconfig. speed**	**Max. reconfig. speed**
OPB_HWICAP (PowerPC cache_disabled)	7.7 KB/12.1 ms	23.2 KB/36.5 ms	44.5 KB/75.6 ms	79.9 KB/135.6 ms	0.61 MB/s	0.64 MB/s
XPS_HWICAP (PowerPC cache_disabled)	7.7 KB/9.2 ms	23.2 KB/27.9 ms	46.5 KB/57.9 ms	80.0 KB/99.7 ms	0.82 MB/s	0.84 MB/s
OPB_HWICAP (PowerPC cache_enabled)	7.7 KB/694.8 μs	22.7 KB/2.3 ms	43.9 KB/4.5 ms	75.9 KB/7.8 ms	10.1 MB/s	11.1 MB/s
XPS_HWICAP (PowerPC cache_enabled)	7.7 KB/336.9 μs	23.2 KB/1.3 ms	44.5 KB/2.5 ms	74.6 KB/4.2 ms	19.1 MB/s	22.9 MB/s
OPB_HWICAP (Microblaze cache_enabled)	7.7 KB/1.3 ms	23.2 KB/3.9 ms	47.1 KB/7.9 ms	77.7 KB/13.0 ms	6.0 MB/s	6.0 MB/s
XPS_HWICAP (Microblaze cache_enabled)	7.7 KB/532.6 μs	23.2 KB/1.6 ms	47.2 KB/3.3 ms	79.1 KB/5.4 ms	14.5 MB/s	14.6 MB/s
DMA_HWICAP	7.7 KB/95.1 μs	23.2 KB/282.3 μs	46.8 KB/566.3 μs	81.9 KB/991.1 μs	82.1 MB/s	82.6 MB/s
MST_HWICAP	7.7 KB/33.0 μs	23.2 KB/98.9 μs	44.8 KB/190.7 μs	76.0 KB/323.1 μs	234.5 MB/s	235.2 MB/s
BRAM_HWICAP	7.7 KB/28.0 μs	23.2 KB/66.3 μs	45.2 KB/121.7 μs	none	332.1 MB/s	371.4 MB/s

Figure 5.8: Reconfiguration speed measurements of ICAP designs for various sizes of partial bitstreams [31]

5.8 Framework FPGA Resources Used

The developed framework has modest resource requirements that leave plenty of resources for user hardware. The number of Flip Flops used is 1245 out of 81920 or 1% of all Flip Flops. The number of LUTs used was 1266 out of a total of 81920 or 1% of the total LUT resources. The number of BRAM resources used was 16 out of 295, or 5% of the total

BRAM resources. Figure 5.9 shows a graphical breakdown of resource usage. The BRAM resources are made up of 8 8KB FIFOS used for TX and RX for four accelerators. This number could vary depending on customizations to the framework.

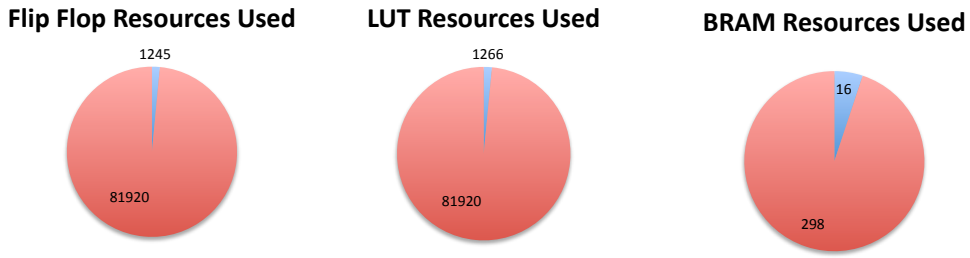

Figure 5.9: FPGA Resources

Chapter 6

Conclusions

The thesis describes a framework for clustering commodity Hybrid FGPAs. The framework supports DMA transfers between the CPU and hardware accelerators by utilizing silicon DMA controllers provided by the Virtex-5 FXT architecture. Runtime reconfiguration of hardware accelerators is supported through the use of partition based partial dynamic reconfiguration. This work expands upon previous works to offer a framework with higher throughput and more flexible accelerator configuration options to increase the range of potential applications.

Angstrom Linux with a kernel from Xilinx was deployed on the FPGA embedded PowerPC processors. Open MPI and gcc development tools were provided to facilitate development on the FPGA. A character device driver for managing data transfers and reconfiguration of hardware accelerators was developed. A procedure on how to implement a hardware accelerator for the framework both in hardware and software was presented. A loopback accelerator case study was performed to detail the design process and collect performance data.

Performance characteristics of the framework were analyzed to identify improvements of previous works, and areas that could be improved further. Throughput between the CPU and hardware accelerator was improved by up to 5.6x using DMA transfers over a previous framework which sent data over the PLB. Partial reconfiguration was show to reconfigure accelerators at a rate of 5MBps. This is a useful metric for future applications which may want to swap accelerators during computation and know going in the overheads associated with doing that.

6.1 Future Work

The framework developed in this thesis provides a foundation for many additional research efforts. This section outlines some of the obvious areas for further research. The first area is in improving the partial dynamic reconfiguration support both at synthesis and runtime.

The developed method of reconfiguring a hardware partition requires an accelerator to be synthesized for the partition desired. Thus an accelerator would have to be implemented for all four partitions in this framework to allow it to be placed in any partition. There is a technique called dynamic bitstream relocation that addresses this issue. It allows a bitstream targeting one specific partition to be modified to work with other partitions assuming they have the same resources. There are software and hardware accelerated relocation efforts [30] which could be applied to this framework. This would reduce the synthesis time required to deploy a new accelerator to a target partition. Another area which could use improvement is the reconfiguration time. The implemented methodology using the Xilinx HWICAP is the most straightforward, however alternative designs methodologies have been explored which use DMA transfers and demonstrate significant throughput improvements [31].

With the ability to dynamically reconfigure accelerators, applications using this feature can be deployed. One area for future study would be to develop an application that would perform certain operations in hardware, then at some later point during the computation swap in different hardware to handle a different phase of the computation. In this respect, different scheduling methodologies could be explored. There are many possible ways that dynamic reconfiguration in a commodity cluster environment could be utilized in future applications.

Another area of interest would be to explore C to HDL tools targeting the developed framework. There are quite a few tools both academic and commercial that perform this translation. It may be desirable to deploy an application targeting this framework using

some exiting or custom tool. This could significantly reduce the application specific hardware design time while still taking advantage of the framework.

Inevitably, the Virtex-5 FXT hardware used in this thesis will become obsolete. In fact, the next Hybrid FPGA architecture from Xilinx called ZYNQ has been announced. Lessons learned from the thesis as well as previous works can be applied to emerging platforms to enable the next generation of Hybrid FPGA clustering. The ZYNQ offers a dual core ARM processor with a high bandwidth interconnect to hardware accelerators and many more features that will enable a new class of applications.

Bibliography

[1] D. E Culler, J. P. Singh *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2005.

[2] J. Mason, "FPGA HPC - The road beyond processors", 2007. *Proceedings of the Third Annual Reconfigurable Systems Summer Institute*, Urbana, Illinois, 18 July 2007.

[3] SRC Computers. SRC-7 Overview, 2009. http://srccomputers.com/products/src7.asp.

[4] Addison Snell, Debra Goldfarb, and Christopher G. Willard. Designed to Scale: The Cray XT5 Family of Supercomputers. Technical report, Tabor Research, Nov 2007. http://www.cray.com/Assets/PDF/products/xt/Tabor_XT5_Whitepaper.pdf.

[5] M. Gokhale, et. al., *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Dordrecht, The Netherlands, 2005.

[6] Jeremy K. Espenshade, "Scalable Framework for Heterogeneous Clustering of Commodity FPGAs," M.S thesis, Rochester Institute of Technology, May 2009.

[7] R. Sass, et. al., "Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing". *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.127-140, 2007.

[8] A. Schmidt, et. al., "Reconfigurable Computing Cluster Project: Phase I Brief", 2008. *16th International Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, CA, pp. 300-301, 14-15, April 2008.

[9] Beowulf.org Overview, 2011. http://www.beowulf.org/overview/index.html.

[10] B. Wilkinson, M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Education, Upper Saddle River, NJ, 2005.

[11] Xilinx Corp. Xilinx Press Release #0204, 2011. http://www.xilinx.com/prs_rls/silicon_vir/0204_v2p_main.html.

[12] Xilinx Corp. Xilinx Press Release #0501, 2011. http://www.xilinx.com/prs_rls/silicon_vir/0501_fx12shipment.htm.

[13] Xilinx Corp. Xilinx Press Release, 2011. http://phx.corporate-ir.net/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1123966&highlight=.

[14] Xilinx Corp. Zync-7000 Extensible Processing Platform, 2011. http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm.

[15] John H. Kelm, "Operating System Interfaces to Reconfigurable Systems," M.S thesis, University of Illinois at Urbana-Champaign, 2006.

[16] J. Jones, M. Stettler, "Dynamic Reconfiguration and Incremental Firmware Development in the Xilinx Virtex 5". *Topical Workshop on Electronics for Particle Physics, Naxos, Greece*, pp.583-586, 15 - 19 Sep 2008.

[17] H. Kalte, M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems". *International Conference on Field Programmable Logic and Applications*, pp. 498-502, 2009.

[18] Xilinx Corp. Appl. Note XAPP1121 - Reference System: Optimizing Performance in PowerPC 440 Processor Systems, October 9, 2008.

[19] Xilinx Corp. Appl. Note XAPP1126 - Reference System: Designing an EDK Custom Peripheral with a LocalLink Interface, December 10, 2008.

[20] Xilinx Corp. Appl. Note XAPP1129 - Integrating an EDK Custom Peripheral with a LocalLink Interface into Linux, May 5, 2009.

[21] Xilinx Corp. DS586 - XPS HWICAP (v3.00a).

[22] Xilinx Corp. Xilinx, DS579 - XPS Central DMA Controller.

[23] Xilinx Corp. Xilinx, DS440 - Channelized Direct Memory Access and Scatter Gather (v1.00a).

[24] Xilinx Corp. Xilinx, UG191 - Virtex-5 FPGA Configuration User Guide.

[25] A. Papkonstantinou et. al, "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs". *IEEE 7th Symposium on Application Specific Processors*, San Francisco, CA, pp.35-42, 27 - 28 Jul 2009.

[26] A. Putnam et. al, ”CHIMPS: A C-LEVEL COMPILATION FLOW FOR HYBRID CPU-FPGA ARCHITECTURES”. *International Conference on Field Programmable Logic and Applications*, Heidelberg, pp.173-178, 8 - 10 Sept 2008.

[27] R. Kallam, ”Application Study of EAPR based Partial Dynamic Reconfiguration”, 2007.

[28] Y. Rajasekhar, et. al., ”FPGA Session Control (FSC): Providing Remote Access to A Cluster of FPGAs”, 2008. *16th International Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, CA, pp.298-299, 14-15 April 2008.

[29] Xilinx, UG702 - Partial Reconfiguration User Guide.

[30] Ramachandra Kallam, ”Accelerated Frame Data Relocation On Xilinx Field Programmable Gate Array,” M.S thesis, Utah State University, 2010.

[31] M. Liu, et al, ”Run-Time Partial Reconfiguration Speed Investigation And Architectural Design Space Exploration”, 2009. *International Conference on Field Programmable Logic and Applications*, Prague, 498-502, Sept 2009.

[32] M. Saldana, H. Hun Liu, ”Using Partial Reconfiguration in an Embedded Message-Passing System”. *2010 International Conference on Reconfigurable Computing*, Quintana Roo, pp.418-423, 13 - 15 Sep 2010.